Validation of Data Flow Results for Program Modules

Dissertation

Schriftliche Arbeit zur Erlangung des akademischen Grades
„Doktor der Naturwissenschaften“
an der Fakultät für Elektrotechnik, Informatik und Mathematik
der Universität Paderborn

vorgelegt von

Karsten Klohs

Paderborn, 2009

Datum der mündlichen Prüfung:
03.04.2009

Gutachter:
Prof. Dr. Uwe Kastens, Universität Paderborn
Prof. Dr. Jens Knoop, Technische Universität Wien

Promotionskommision:
Prof. Dr. Uwe Kastens, Universität Paderborn
Prof. Dr. Jens Knoop, Technische Universität Wien
Prof. Dr. Heike Wehrheim, Universität Paderborn
Prof. Dr. Heiko Platzner, Universität Paderborn
Dr. Mathias Fischer, Universität Paderborn

Abstract

This thesis presents a general approach to the *validation* of *interprocedural data flow results* for separated *software modules*, in order to enable the safe use of data flow results on devices which cannot afford to run the data flow analysis on their own. The underlying idea stems from the "Proof-Carrying-Code Principle" [Nec97], which utilises that it is easier to check the correctness of a given solution of a problem than to solve the problem.

The requirement to validate analysis results originally arose for Java Bytecode Verification on Smart Cards. The generalisation of this specific application to the validation of interprocedural data flow results enables advanced optimisations or security checks on limited devices in a scenario where the mobile code is transmitted via an inherently insecure transport media like the Internet. The validation ensures the correctness of the results but the code producer can perform the complex analysis on a more powerful machine.

The central contribution of this thesis is the extension of the validation approach to the interprocedural analyses and to separated software modules. This is vital in a mobile code scenario where different software modules can be dynamically loaded to the target device and where the potential interactions between the software modules and the runtime environment have to be considered.

Zusammenfassung

Diese Arbeit beschreibt einen allgemeinen Ansatz zur *Validierung* von *interprozeduralen Analyseergebnissen* für einzelne *Softwaremodule*, um die sichere Nutzung von Datenflussergebnissen auf Zielplattformen zu ermöglichen, die die Analyse nicht eigenständig durchführen können. Die zugrunde liegende Idee entstammt der "Proof-Carrying Code"-Methodik [Nec97], die sich zu Nutze macht, dass es einfacher ist, die Korrektheit der Lösung eines Problems zu überprüfen als das eigentliche Problem zu lösen.

Die Notwendigkeit, Datenflussergebnisse zu prüfen, entstand ursprünglich bei der Java Bytecode Verfikation auf Smard Cards. Die Verallgemeinerung dieses speziellen Ansatzes auf die Validierung von interprozeduralen Analyseergebnissen ermöglicht erweiterte Optimierungen oder Sicherheitsüberprüfungen in einem Umfeld in dem mobiler Code über ein unsicheres Transportmedium wie dem Internet übertragen wird. Die Validierung stellt die Korrektheit der Analyseergebnisse sicher, aber der Codeerzeuger kann die komplexe Analyse auf einer leistungsfähigeren Maschine durchführen.

Der wesentliche Beitrag dieser Arbeit ist die Erweiterung des Validierungsansatzes auf interprozedurale Analysen und auf die Analyse einzelner Softwaremodule. Dies ist entscheidend in einem Umfeld, in dem verschiedene Softwaremodule zur Laufzeit auf eine Zielplattform geladen werden können und wo die möglichen Wechselwirkungen zwischen Softwaremodulen und der Laufzeitumgebung berücksichtigt werden müssen.

Acknowledgements

First of all, I wish to thank Uwe Kastens, my advisor. His keen sense of interesting directions of research and his stance on science in general has shaped this thesis - and me - in many ways. The freedom of research is something which I learned to appreciate progressively during the years. His comments on the early versions of this thesis had sometimes been extensive, but always constructive and helpful.

I am also grateful to Jens Knoop who has offered me the opportunity to discuss the fundamental concepts of my thesis with a broader audience. I'll always remember the hospitality I experienced in Vienna and the admirable precision with which Jens Knoop is able to pinpoint the challenging parts of a problem.

Additionally, I'd also like to thank my colleagues for many interesting discussions - especially Michael Thies for his remarkable ability to give me the feeling that at least someone understands nascent ideas even before they have been fully developed.

However, I am most grateful to my beloved wife Monika - her calm and serene support kept me grounded even in the stressful phases of the thesis.

Contents

1 Introduction

This thesis presents a general approach to the *validation* of *interprocedural data flow results* for separated *software modules*, in order to enable the safe use of data flow results on devices which cannot afford to run the data flow analysis on their own. The central contribution is a reconsideration the generic functional approach to interprocedural analysis in the validation scenario and an adoption of the approach so that it can deal with analysis results of software modules which are analysed in isolation.

The validation of interprocedural analysis results is attractive because an efficient validation process can still meet the resource-constraints of a limited target device, while the consideration of interprocedural data flow significantly extends the expressiveness of the framework. However, a mobile code scenario where additional code can be dynamically loaded on a target platform at runtime implies that a code producer of a single software module does not know all the code which comes to execution on the target device. Therefore, it is vital that the analysis framework supplies support for a modular analysis which considers the potential interactions between software modules even if each single software module is analysed in isolation.

From a more general perspective, the validation of data flow results is an application of the proof-carrying code principle to protect a target device from potentially malicious effects of mobile code. In his original work Necula [Nec97] attaches a proof to the code of a device driver to ensure that it is safe to load the device driver into the kernel memory. The approach exploits that it is easier to check the correctness of a proof than to construct the proof. Similarly, the check that a given result actually solves a data flow problem is simpler than to solve the analysis problem.

The Proof-Carrying Code principle is an interesting approach to mobile code safety because the code consumer is sure that the solution is correct and the validation costs are limited to the load time of the code and do not impact the runtime efficiency of the program. Other approaches protect the integrity of the target device in a different way. One option is to execute mobile code in a secure sandbox which requires that special runtime checks ensure that the program behaves well. In this scenario the consumer does not have to trust the code producer but the runtime checks impact the runtime efficiency of the program. Another option is to attach a digital signature to the mobile code and the result which ensure that neither the code nor the result have been manipulated during transition. The check of the digital signature produces almost no costs on the target device. However, the use of digital signatures requires a key-exchange

protocol and the code consumer has to rely on the fact that the producer has solved the problem correctly.

Java Bytecode Verification on Smart Cards is a classical problem which can nicely be solved with the proof-carrying code principle but which is difficult to tackle with the other techniques. The most challenging part of the Bytecode Verification is to solve an intraprocedural type inference problem which ensures that the program is type safe. This data flow problem cannot be solved on a Smart Card but it is possible to validate a given solution of the type inference problem. In contrast, it is prohibitively costly to enforce the type safety of a program by runtime checks for each executed bytecode instruction in a sandbox. Similarly, digital signatures cannot ensure that the code producer has performed the type checking after the generation of the software because Java code which is transmitted to a virtual machine can even stem from a completely unknown source.

The type inference problem of the Java Bytecode Verification is an intraprocedural data flow problem. Furthermore, Rose [Ros03], and Albert [APH04] observe that data flow problems fit into the proof-carrying code methodology because a given data flow result can be checked by showing that it solves the system of data flow equations which specify the problem. Data flow analyses form an attractive problem class because they are based on a well understood theory which originates in the works of Kam, Ullman [KU76], [KU77], and the abstract interpretation model of Cousot [CC77]. Furthermore, the general framework has been applied to numerous problems which span a large design space with different trade-offs between expressiveness and efficiency.

The central contribution of this thesis is the reconsideration of the validation approach in the interprocedural setting and the support for an analysis of separated software modules to capture the potential interactions between the runtime environment and different pieces of mobile code. To achieve this, we develop a model for the validation of interprocedural results which can be applied to an interesting class of analysis problems in a uniform way. The essential part of the model is a uniform representation of interprocedural analysis results which supports all operations required for the validation. This disburdens the developer of the analysis from the effort to specify a problem-specific validation technique and a result representation for each analysis. Furthermore, the model integrates support for dynamic method binding, different strategies to deal with external code, and normalisation techniques which reduce the size of the result representation into the validation process.

1.1 Methodical Contributions

On the way to the solution of the central goal of this thesis we have to reconsider many aspects of traditional data flow analysis techniques in the validation scenario. This yields methodical contributions in the following areas:

Validation of Interprocedural Results Our starting point is the functional approach to interprocedural analysis [SP81] and the observation that the validation of a data flow corresponds to the check that the given result solves the system of data flow equations which specify the analysis problem [Ros03], [APH04].

In order to facilitate the validation of interprocedural analysis results we reinterpret the general validation principle in the summary function model. Vital is the key observation that the functional approach formulates the computation of summary functions also in terms of a data flow problem.

However, the complexity of the underlying system of data flow equations increases because it encodes the interprocedural flow graph, which takes the summary functions of the callees at a call site into account. Furthermore, the equation system which is to be checked by the validator deals with summary functions and not with data flow values. As a consequence, the validator has to be capable to compare the summary functions given in the certificate and the summary function which describe the requirements of the program with each other efficiently.

However, the generic functional approach does not make any assumptions about the representation of the summary functions, so that the problem of an efficient comparability of summary functions would have to be resolved for each new analysis problem. Thus, we develop a summary function representation which can represent an interesting class of analysis problem in a *uniform* way and which supplies all operations which are required for the validation process.

A Generic Model The specification of a generic summary function which meets the requirements of the validation process is the core contribution of the thesis. Essentially, the summary function model solves three different problems:

1. The model abstracts from a concrete analysis because the model reduces the specification effort of a data flow analysis to the specification of a suitable *inducing data flow lattice* and the specification of *instruction-level summary functions*. The functional approach solves the interprocedural aspects of the data flow problem in a generic way based on this input.
2. The validation process requires that it is easy to compare summary function representations in the transmitted result to summary functions computed during validation. We achieve this, by the specification of normalisation rules. The rules yield function representations that can be compared easily by the comparison of their internal structure.
3. The second goal of the normalisation is to reduce the memory requirements of the function representation. Essentially, the normalisation rules can be interpreted as a partial evaluation mechanism which operates on constant elements of the inducing data flow lattice.

The function model is an adoption and reformulation of other generic interprocedural analysis frameworks like the ones of Reps et al. [RHS95], [SRH96]

and Knoop [Kno99], to meet the requirements of the validation process. The model of Reps defines a generic function representation which uses a decomposition of the program state into an tuple of data flow values, a bipartite graph to model summary functions, microtransformer to express properties of the inducing data flow problem, and an interprocedural flow graph to integrate the summary functions of callees into the summary functions of the caller. The substantially new achievement of the function model in this thesis is that the function model combines the different aspects in a single function representation and that this representation directly meets the requirements of the validation process.

Representation of Modular Results The summary function model does not only supply the basic infrastructure for the validation process but it can also be extended for the representation of results which stem from the analysis of a separated software module.

The functional approach to interprocedural analysis computes a summary function for each method which comprises the effects of an invocation of the method with respect to the problem. If a software module is analysed in isolation, then summary functions which model the effects of methods outside the module are not available.

We model this situation by the introduction of *function variables* in the summary function representation. Function variables act as placeholders for currently unavailable summary functions. The function variables represent the potential influence of external code on the analysis result of the software module but it does not resolve this dependency. Therefore, it is possible to use the function variables in two different ways. Firstly, it is possible to integrate results of other software modules later. Secondly, it is possible to replace the function variables by safe assumptions about the behaviour of the external code. With the first technique we can combine the results of different software modules, while the second technique corresponds to an isolated analysis of a single software module where the assumptions about external code is explicitly encoded in summary functions which replace the function functions.

This modelling technique is novel, because it integrates the potential impact of the behaviour of other software modules directly into the summary function representation. This is a difference to other approaches like the component-level analysis of Rountev [RSX08] which uses a separated flow graph model to deal with external method invocations. The direct integration is advantageous, because function variables are also subject to the normalisation process. This way, it is possible to rule out dependencies on other software modules which do not influence the analysis result. Even more importantly the integration of summary function variables into the function model is compatible with the validation process. Thus, it is also possible to validate such an open representation where the dependencies on external code have not yet been treated.

Dynamic Method Binding and Class Loading The resolution of dynamically bound method calls is a prerequisite for any interprocedural analysis of an object-oriented program. The target of a dynamic call depends on the runtime type - or more precisely on the runtime class - of the object the receiver reference points to. Therefore, a static program analysis has to find a safe approximation of the set of classes of all potential receiver objects. This set defines the set of potential call targets whose summary functions have to be considered at the call site.

The determination of the potential classes of receiver references in a runtime environment which permits dynamic class loading is a challenging task. The reason is that it is no longer possible to treat the dynamic calls just by an inspection of their declared type only. The declared type of the receiver reference implicitly includes all subclasses so that the analysis has to assume that additional method implementations are contributed by dynamically loaded classes at virtually any dynamic call site. Usually, the analysis can make very conservative assumptions about such unknown method implementations, which significantly reduces the precision of the analysis.

The problem can be approached from two directions. Firstly, we can choose an intermediate way between the safe but very restrictive worst-case assumption, that any class can be subclassed and the overly optimistic closed-world assumption that the whole program is known, and no class can be further subclassed. The *closed-program assumption* expects that all but the classes of the software module under consideration can still be extended by additional subclasses. This is reasonable for applications which stem from a vendor who does not intend to modify the program after deployment.

However, this strategy cannot be applied for libraries and frameworks because they are designed for being extended. Therefore, we specify a data-flow based type inference algorithm which tries to determine the potential classes a receiver reference of a dynamic call may point to exactly. The key observation is that the class of an object instance is known exactly immediately after the instantiation of the object. We capture this intuition in terms of so-called *point types* each of which represents instances of a specific class in a precise type model. A type inference analysis can use this model to detect the set of potential classes of the receiver objects of a specific call.

This analysis technique is a variant of existing type inference analyses. The main contribution of this thesis, is to show how we can specify a type inference algorithm in terms of the *validatable* summary function model. As a consequence, the type results can be checked by the validation techniques developed in the thesis. This leads to validatable interprocedural flow graphs even in the presence of dynamic method binding and a dynamic class loading mechanism.

Optimisations of the Validation Process Several optimisation strategies have been proposed to improve the efficiency of the validation process [BLTY03], [RR98], [KK05] in the intraprocedural setting. We reinterpret such techniques in

the interprocedural scenario in two steps. Firstly, we abstract from the problem-specific details of the intraprocedural formulations. Secondly, we show how the techniques can be applied to the generic summary function model.

The second step provides additional insights about the interprocedural validation scenario. For example, one of the most promising optimisation ideas is the difference certificate approach which originates in Rose's approach to lightweight bytecode verification [RR98], [Ros03]. The idea is to ship only information in the certificate which represents the differences between the data flow values computed during the validation pass for checking purposes and the final analysis result. The reinterpretation of this strategy in the interprocedural scenario reveals that it is necessary to derive *difference functions* in order to apply the approach within the summary function model. Fortunately, the summary function model turns out to meet this requirement.

1.2 Limitations

The model which is developed in this thesis for the validation of interprocedural analysis results of software module deals with many aspects of the interprocedural analysis of object-oriented programs and enables the validation of the analysis results.

However, the current prototype implementation and the evaluation instantiate the whole framework with a comparatively simple set of module implementations. Currently, the framework suffers from the following limitations.

Program State The environment which represents the program state at a program point contains local variables, parameters and result values of method invocations only. In other words, the framework tracks the data flow through the call stack of the program only. This is sufficient to define interprocedural variants of intraprocedural analyses which operate on the local variables and simple escape analyses. The extension to global fields is straight forward, because it requires the introduction of a new data flow variable for each field only. In contrast, we expect that the consideration of the data flow via the object heap is a challenging task because it may require support from a limited points-to or alias analysis to identify the accessed object fields more precisely.

Inducing Data Flow Problems The evaluation of the prototype implementation uses a simple copy constant propagation as an inducing analysis and shows that the validation approach is suitable in this scenario. Other inducing analyses are specified in Chapter 7 but they are not completely implemented and evaluated yet. However, general considerations justify the claim, that all data flow problems which are efficiently representable IDE problems in the sense of Reps [RHS95] are suitable targets of the validation approach as well.

Interprocedural Precision One result of the evaluation is that more than 20% of the data flow values depend on interprocedural data flow even though the framework immediately uses pessimistic assumptions if it encounters language constructs like field accesses. Thus, an interprocedural analysis approach is promising in general, because it can determine more precise data flow information for a significant amount of data flow values even in the current implementation. However, the copy constant propagation turns out to compute pessimistic values for almost all values which depend on interprocedural data flow. This is due to the fact, that interprocedural dependencies are currently restricted to result values of method invocations - which usually not return known values. We expect that other analyses like the type inference analysis specified in Chapter 7 exhibit a better interprocedural precision improvement, for example if the analysed software contains factory methods, which return references of a specific class.

Validation Scenario The framework implements the simplest validation scenario, where the producer analyses a software module in isolation, treats all external dependencies according to a specific strategy and ships the closed result to the validator in a complete certificate. However, the validator is also already capable to validate an open result representation which still contains external dependencies. Furthermore, the evaluation shows that this more complex validation is manageable. Nevertheless, the proper use of this open result implementation in an incremental or partial analysis scenario as well as the application of optimisations strategies as discussed in Chapter 6 remain to be fully implemented.

The Value Computation in a Modular Setting The effectiveness of the value computation phase in the interprocedural approach depends strongly on the potential entry points into the analysed software. This is usually not an issue for a whole program analysis because it expects the main method of the program as single entry point and it usually rules out potential call-backs into the program e.g. by system calls. However, the question is an important issue for a single software module. Usually, a module is intended to interact with other modules, so that all methods of the method can be entry points at the first glance. We intentionally defer the development of strategies for the restriction of entry points into a module, because interestingly the evaluation reveals that the functional part of the analysis already yields a significant amount of analysis information.

1.3 Road Map

The thesis is mainly structured according to the methodical contributions summarised in the previous section. Chapter 2 provides an overview about several application scenarios for the validation techniques developed in this thesis. Chapter 3 summarises the properties of the traditional data flow analysis

framework and establishes the basic terminology. Furthermore, we consider the relationship between data flow analysis and model-checking techniques in order to figure out potential effects of data flow validation techniques in a larger context.

Chapter 4 formulates the general validation principles and reinterprets them in the interprocedural setting. The *general validation principle* states that the validation of data flow analysis results corresponds to the check that a given data flow result solves the system of data flow equations. Furthermore, the validation pass can validate any valid solution for the equation system. Thus, it is possible to weaken the analysis results as long as they remain a solution of the data flow problem. The analysis phase can apply this *intentional under-approximation principle* to improve the efficiency of the validation pass. The reinterpretation of these principles for the functional approach leads to the *interprocedural validation principle*. Essentially, the underlying system of data flow equations gets more complex in the interprocedural case but the general principles can still be applied. Finally, the *safe-approximation principle* states that it is possible to approximate the solution if we replace all variables in the equations by safe lower bounds. This is vital to deal with data flow values which depend on the behaviour of external modules.

Chapter 5 contains the essential contributions of the thesis. It develops the summary function model for the validation of interprocedural results. The model reduces the function representation to normal forms which can be compared to each other on a structural level. This keeps the summary function model generic and increases the efficiency of the representation. An additional contribution is that the model represents dependencies on other software modules explicitly in terms of function variables so that the dependencies can either be replaced by safe assumptions if the module is considered in isolation or more precise results for other modules can be integrated later.

Chapter 6 reconsiders different optimisation strategies for the validation process in the interprocedural setting. The goal is to figure out, how the optimisations can be applied to the validation of the functional result of the interprocedural analysis.

A discussion of different program analyses in Chapter 7 serves several purposes. Firstly, it shows how different analyses can be specified in terms of the generic model. Furthermore, we investigate how different characteristics of the inducing analysis influence the complexity of summary function representation. Finally, we consider the specification for a type inference algorithm in terms of the summary function model to highlight the impact of open class hierarchies on the interprocedural analysis scenario.

The following chapter contains an overview about the current state of the prototype implementation of the analysis framework. The description focuses on the structure of the framework and explains how the different kinds of modules in the framework are currently instantiated.

The evaluation of the system investigates a full certificate approach which instantiates the generic framework with a simple copy constant propagation as

an inducing analysis and yields several results. Firstly, the measurements show that an interprocedural analysis is promising because more than 20 % of the analysis result depends on interprocedural data flow even for the comparatively simple instantiation of the framework. Unfortunately, the example analysis does exploit this potential because interprocedural copy constants are very rare in the subject software. Secondly, we investigate the internal structure of the function representation to show that the memory requirements for the certificate and during the validation process remain manageable. Finally, a runtime comparison of the analysis and the validation phase reveals that the linear pass of the validation is in fact significantly faster than the iterative analysis.

2 Application Scenarios

The validation of analysis results is a useful approach to tackle different kinds of problems. Firstly, we describe the basic application scenario in this chapter in order to emphasise the characteristic properties which call for a validation approach. Secondly, we discuss several concrete application scenarios which fit into this general setting.

The key observation which forms the starting point of the whole approach is that it is often much more efficient to validate that a given solution solves a specific problem than to compute the solution. At the same time the validation process ensures that the solution is correct, so that it is not necessary to ultimately trust the computation phase.

This general principle can be used to separate a code producer from a code consumer in a safe way. In this thesis we apply this general principle to the validation of data flow results. Consider the situation in Figure 2.1. In the

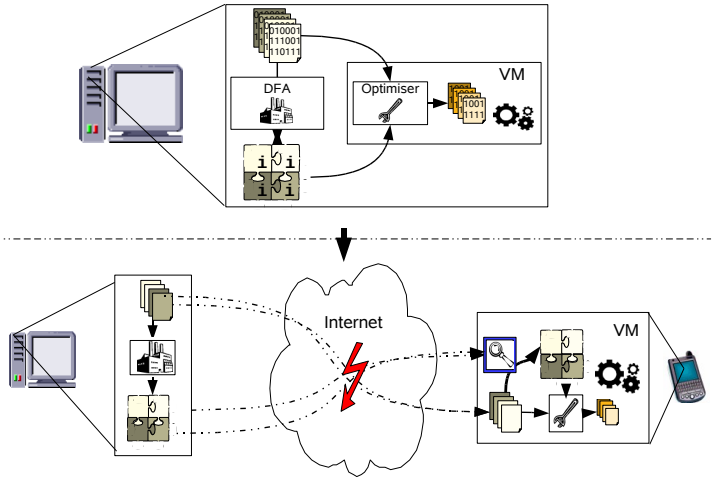


Figure 2.1: Validation of Analysis Results

traditional setting data flow analysis results are computed and immediately used on the same host. For example a program optimiser can use the results of a constant propagation or type inference analysis to produce an optimised version of the program.

An important problem arises if the use of erroneous results has the potential to break the integrity of the consumer. In this case, the consumer has to check that

the analysis results correspond to the given program. Digital signatures are not an option if the code consumer does not trust the producer or if key exchange protocols cannot be established. In such a situation, the validation of the data flow results is an interesting solution, because it checks that the analysis results are correct but at the same time it is less costly than the computation of the analysis results.

Java Bytecode verification is an example for the validation of data flow results. Essentially, the Java virtual machine checks that the program in question is type safe which can be expressed as the result of an intraprocedural type inference problem. The computation of the type values has been too costly on very restricted devices like smart cards but the validation of given results is possible in such target environments. The check that a given program is type-safe is vital to protect the virtual machine against several low-level attacks like the manipulation of reference values which can be used to bypass other security mechanism like security managers etc.

All in all, the validation of data flow results is useful in application scenarios which separate the analysis phase from the usage of the results and which exhibit the following properties:

Different Computational Capabilities The code consumer has only limited computational capabilities at his disposal so that he cannot perform the analysis on its own. Thus, *efficiency* is mandatory for all operations performed at the consumer site.

Untrusted Producer The code consumer cannot ultimately trust the code producer. Thus, a *validation* of the given results is required to protect the code consumer against the use of erroneous results.

This thesis focuses on the validation of interprocedural analysis results, because they are a good choice for the intended application scenario. On the one hand, a rich number of program properties can be computed by interprocedural analyses which range from simple ones like the determination of constant arithmetic values to complex points-to analyses. Interprocedural analyses have the potential to be more expressive than their intraprocedural counter parts because they are able to track the data flow across the boundaries of method calls.

On the other hand, data flow analyses are very efficient compared to more sophisticated approaches like model-checking. Model checkers can check much more precise properties but they suffer from the problem that the considered state space of the program grows rapidly. Thus, model checkers are only suitable to deal with programs of a limited size. In contrast, data flow analyses can be performed efficiently even on large program modules. Thus, interprocedural data flow analyses are a reasonable choice for the given application domain.

We will now consider different kinds of application scenarios which benefit from the validation of analysis results.

2.1 Security Policies and Mobile Code

The idea to validate given information about a program was originally applied in a security scenario. The original proof-carrying code approach [Nec97] formulated the fact that driver programs do not perform arbitrary accesses to the kernel memory in terms of a proof in first order logic [NL96]. After the consumer has checked this proof it is guaranteed that the given driver program can be safely integrated into an operating system kernel.

In this scenario the validation of the proof is an end in itself, because it immediately guarantees that the program exhibits the properties that the consumer wishes to enforce. An important observation is that the kinds of properties which can be checked depend on the calculus the proof is specified in. The more expressive the underlying calculus is the more program properties are captured. However, more expressive calculi are usually more costly to check. Therefore, a reasonable trade-off has to be chosen according to the application scenario.

A closely related observation is that a limited device that is deployed in a network environment has to restrict itself to the validation of moderate program properties. Full-fledged program verification techniques are usually prohibitively costly to check, even with the support of producer supplied annotations.

Data flow analysis problems constitute an interesting calculus in the security scenario. Firstly, there exists a rich set of interesting data flow problems which have been used in a number of different application domains already. Thus, the number of program properties which can be expressed by data flow results is significantly large. Secondly, the abstractions that model a data flow problem are usually closer to the code under consideration than properties of the high-level model of the software system. This is important because in the end the code itself is the final instance which specifies the behaviour of the program. Finally, it is often easy to specialise data flow analyses so that they express additional properties of the code.

A very prominent example of the application of data flow techniques are static type systems. Statically typed languages impose restrictions on the programmer but in turn they provide guarantees about the behaviour of well-typed programs. Type checking identifies errors early which significantly increases the robustness and maintainability of the software product. The type checking mechanism involves data flow techniques. For example, the most complex part of the Java Bytecode verification solves an intraprocedural type inference problem to establish the type correctness of the program. One of the major advantages for the validation of type annotations of a program is that the type checking is done by the consumer immediately before the code comes to execution. This is important if the consumer wants to enforce the well-typedness of the program on its own.

Furthermore, type inference techniques can be extended so that the constructor of the consumer device can specify and communicate security relevant proper-

ties by additional type constraints. A simple example is the introduction of a `null`-type into the type inference algorithm which enables the type analysis to conclude whether a reference type can be null or not. This is useful to enforce that parameters of some security relevant method call are never `null` or to remove redundant checks from the code. So called type annotations [PACJ+08] which head into this direction are even going to be integrated into the next version of the Java language.

All in all, the use of data flow analysis results to specify security relevant properties of the program is attractive. The wide adoption of the techniques and their efficiency can be more relevant arguments than the fact than conservative nature of data flow analyses restricts their expressiveness.

2.2 Program Optimisation and Partial Analyses

The validation of the results of data flow analyses is even more attractive if the results are used for program optimisations which is their original field of application. The most important advantage is that existing definitions of data flow analyses can be used immediately to trigger optimisations at the consumer side.

However, the optimisations can compromise the integrity of the consumer if they are based on erroneous data flow information. For example, a dynamically bound method call can be bound statically if the runtime type of the receiver reference can be determined exactly by a static analysis. If the wrong receiver type is given, then the optimisation will bind the method invocation to the wrong method implementation. This may lead to malicious memory accesses, for example if a method implementation of a subclass that operates on additional fields is executed on an instance of the superclass that has a smaller memory layout. Therefore, the validation of given data flow results is a major concern, even if they do not describe security properties directly.

The application scenario has but another interesting property which distinguishes it from the security scenario: The abandonment of a specific optimisation does not break the integrity of the consumer. This offers an additional degree of freedom for the validation process. The producer can omit data flow results which do not lead to optimisation opportunities or which become to costly to be validated by the consumer.

The general idea is that the consumer can focus on the validation the *useful* data flow information only. The loss of some optimisation opportunities may very well be acceptable if in turn the costs of the validation process can be adapted to the capabilities of the consumer. The producer can apply this principle in its full strength because additional efforts can be spend at the producer site to determine the best trade of between precision and validation costs beforehand. Furthermore, even the consumer can apply the principle to protect himself from denial of service attacks: whenever the validation of specific parts of the data

flow result becomes too complex, the validator is free to drop optimisation opportunities.

The application of optimisations at the consumer side has additional advantages than just the protection of the consumer against erroneous optimisations. Many optimisations require runtime support by the consumer and cannot be performed by the producer, if the final target platform of the code is not known. For example, the static binding of methods has to be supported by the virtual machine and requires explicit knowledge about the memory layout of the virtual method tables. This information depends on the implementation of the target virtual machine, so that the application of the optimisation has to be deferred until the code has been transmitted to a concrete target platform.

2.3 Modular Results and Partial Analysis

The capability to deal with data flow results of software modules separately is useful in many ways. Our goal is to define modular analysis results in a way, that they still exhibit the potential dependencies on other modules. The advantage of such a result representation is that we can treat the dependencies on other modules in a flexible way. This allows for the support of more sophisticated application scenarios.

We can already take advantage of a modular result representation at the producer side. Every program uses some sort of interface for example to access the low-level IO-mechanisms of the operating system. Furthermore, each program or module can interact with other programs in various way. If a modular result representation captures such dependencies *explicitly*, then we can estimate the potential effects in different ways, which is depicted in Figure 2.2. For example, a very common technique is to apply the "closed-world" assumption. The analysis phase derives its results and implicitly expects that the program under consideration will not be extended. This is an optimistic assumption, because most runtime systems allow for the late integration of additional plugins or classes. Thus, another way to deal with the potential effects of software infrastructure on the target platform is to treat them completely pessimistically. This is safe but has the potential to loose significant precision. A modular result representation enables the producer to apply one of the strategies or even more sophisticated ones depending on the application scenario. Obviously, the code consumer has to use the same strategy to validate a specific result variant. Thus, the flexibility of a modular result representation can be used to adopt the analysis and the validation phase to different application scenarios easily. In this thesis we will use this technique in the evaluation to compare the potential effects of different approximation strategies with each other.

The second advantage of a modular result representation is that it enables an *incremental validation scenario* as depicted in Figure 2.3. Each larger software module like an application program inherently consists of smaller modules like packages and single classes. Thus, a modular result representation is able to

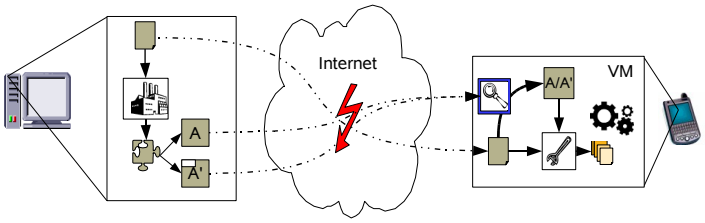


Figure 2.2: Using Modular Results by the Producer

express the results for such smaller software components individually. Thus, the validation process can start validation even before the whole program is transmitted. If it is possible to validate some of the results, then the validator can already use these pieces of the result to apply optimisations ahead of time. Furthermore, the validator can drop pieces of the results as soon as they are no longer needed for the validation of the remaining parts of the software.

Essentially, this scenario calls for two additional capabilities on the target platform. Firstly, the validator has to be able to treat potential effects of missing software pessimistically. Secondly, the validator also has to be able to validate the modular result representation which is now subject to the validation process. We show in this thesis that it is in fact possible to validate the modular result representation defined in our framework and the implementation is already able to apply this principle.

Finally, the modular result representation also provides the basic infrastructure to extend the system to a *partial validation scenario*. Assume that two software modules are analysed separately either on different platforms or at different points in time. The capability of the validator to validate a modular result allows for a validation of the results and a late integration to a complete result at the consumer side as depicted in Figure 2.4. The differences to the incremental scenario are subtle but important. In the incremental scenario we expect that the producer has knowledge about the whole program in question. Thus, the analysis phase can compute the final result for the whole program. Therefore, it is possible to add additional pieces of information about the remaining software components together with the modular results of the first components. Such pieces of information support the validator during the validation process and during the construction of the final analysis results. Furthermore, the analysis

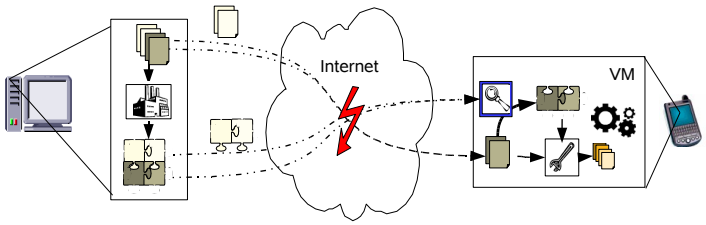


Figure 2.3: Incremental Validation Scenario

phase is able to resolve cyclic dependencies between the different software modules by a fix-point iteration which yields a precise result.

In contrast, we expect that the analysis phase in the partial validation scenario is not aware of the whole program. Although the validator is still able to validate the modular results, it now lacks the support for the validation and composition of the results. Most importantly, the analysis phase is not able to resolve cyclic dependencies between different modules anymore. Nevertheless, it is still possible to resolve cyclic dependencies within a single module. The partial analysis scenario requires that the validator is able to compose modular results and that cyclic dependencies between software modules are treated conservatively. The model which we present in this thesis supplies the required infrastructure, but we will not investigate this rather complex application scenario in detail.

All in all, the capability to validate a modular result representation is one of the core challenges which has to be solved to adopt the framework to various realistic but sophisticated application scenarios.

2.4 Validation of Data Flow Results as an Assisting Technique

There also exist additional application scenarios for the validation of data flow results where the validation process is not an end in itself.

An interesting idea is to combine the validation of data flow results with other validation techniques. This way, the efficiency of the data flow analysis and the increased expressiveness of other approaches can benefit from each other.

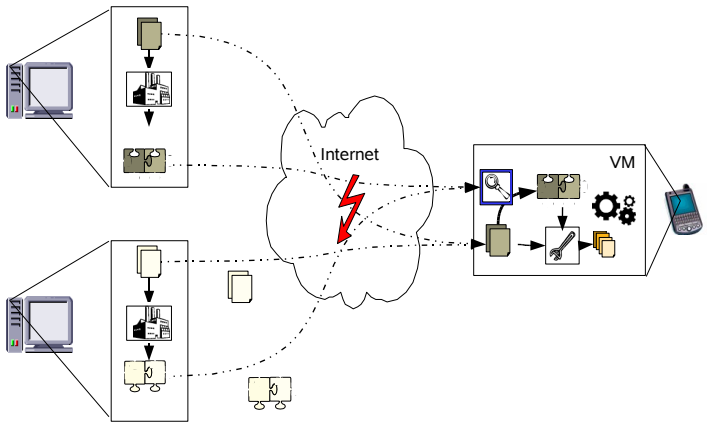


Figure 2.4: Partial Validation

Model checkers can validate program properties expressed in temporal logic. Formulae in temporal logic define program properties in terms of logical combinators, atomic propositions and quantifiers over execution paths of the program. Atomic propositions are very basic ones like "variable x has value 2" or "variable x has the same value than variable y". As a consequence, the state space of a model checker which tries to prove a given formula grows rapidly because it has to represent a huge amount of different program states on different program paths.

The results of a data flow analysis can be used to reduce the state space of the model-checker significantly. For example, the data flow result that a given variable always holds a positive value rules out the half of the potential values which have to be considered by the model-checker.

The usual argument why all properties of the program should be checked by the model-checker is that this reduces the base of trust to the implementation of the model-checker. Obviously, it is an advantage if the consumer which wants to check specific program properties has to rely on a small code base only, because errors in the implementation of the validation phase reduce its value significantly.

However, the combination with data flow techniques is an interesting choice, because data flow problems have a well established formal definition which leads to a generic framework which can be instantiated for several analysis easily. The validation of data flow results reduces the code base even further, because the iterative fix-point computations do not have to be trusted anymore. The validation proves that the solution is a valid solution of the data flow problem in question whether or not the implementation of the analysis phase is correct or not. Thus, the results can be safely used to streamline a subsequent

model-checking phase, so that it becomes applicable on the limited target platform. The important observation is that only the validation pass has to be trusted additionally.

The validation of analysis results can also be interesting for approaches which aim at *translation validation*. The goal in this scenario is to show that the translation of a program by a compiler has preserved the semantics of the program. The transformations applied by the compiler often depend on the results of data flow analyses. This piece of the translation process can be validated with the techniques presented in this thesis. As a consequence, the validator can convince himself that the data flow analysis phase has operated correctly for the program in question even though the implementation of the analysis phase may still not match its specification completely.

This observation gives also rise to another application of the validation techniques for data flow results which supported the implementation of our own analysis framework: The validation pass can detect potential errors in the implementation of the fix-point algorithm which solves the analysis problem. This supports the *implementation of an analysis* and helps to increase the robustness of the whole framework. For example, the validation revealed subtle errors in a caching mechanism for instruction-level summary functions or in the lookup-procedure for dynamically bound method calls whose occurrence depends on the sequence in which the iterative algorithm processed intraprocedural control flow nodes. Furthermore, the generic nature of our approach supplies a rich set of basic datastructures for the implementation of validatable analysis which significantly decreases the size of the additional code base required for the implementation of a new analysis.

To summarise, it is possible to embed the the validation of data flow results in other larger application scenarios as well. Therefore, it is interesting to study the underlying principles even from a more general perspective than in our main scenario.

3 Foundations

3.1 Iterative Data Flow Analysis and Equation Systems

This section summarises the elements of the traditional data flow analysis framework and its relationship to equation systems. The traditional framework is defined with respect to the flow graph of the program which captures the flow of control in the program and the potential execution paths. Data flow analysis computes information about the program state for each program point by an iterative algorithm which propagates data flow information through the flow graph.

The flow graph model and the iterative solution algorithm are closely related to an equation system and the determination of a valid solution for this system. The system of data flow equations highlights the structure and the interdependencies between data flow facts and is especially useful to explain the fundamental principles of the validation of data flow analysis results in Chapter 4.

Therefore, this section briefly reviews the traditional model for data flow analysis to establish the basic terminology and to emphasise the most important properties of data flow analyses which will be reconsidered in the validation scenario later on.

The original formulation originates in the work of Kam, Ullman, and Kildall [KU76], [KU77], [Kil73]. Introductory presentations can be found in any compiler text book [Hec77], [Muc97], [ALSU07], and a comprehensive survey is given in [MR90].

The close relationship to equation systems and Gaussian elimination techniques forms the foundation of elimination algorithms for data flow analysis [RP86].

3.1.1 Elements of Data Flow Problems

The traditional model defines a data flow problem D as a quadruple $\langle G, [\![]\!], L, T \rangle$ where G is a flow graph, $[\![]\!]$ is the so called label function, L is a lattice and T is the function space of transfer functions.

The flow graph G models the flow of control in the program. If the algorithm performs intraprocedural analysis this is the loop and branch structure of the method. Interprocedural analysis extends the flow graph with the call graph of the program, which describes the calling relations between the method implementations in a program.

The lattice L models data flow information we are interested in. We denote the elements of the data flow lattice as *data flow facts*. They represent some assertions about the program state which hold for every possible execution path at a specific point. The information can range from simple yes-no statements, e.g. the information whether the value of an arithmetic expression is available at a program point, to more complex information like static types of local variables.

The set of transfer functions T models in a quite general sense the semantics of the program with respect to the data flow problem in question. A transfer function describes how the execution of the corresponding program fragment modifies data flow facts which model the information about the program state.

The lattice L and the transfer functions in T model the program independent part of the analysis problem. Therefore, they are often called the *data flow framework*. In contrast the flow graph G represents a specific program, for which the data flow analysis is to be performed. The label function M connects the program and the framework. It assigns transfer functions to each piece of code - usually to each node of the flow graph.

A solution of a data flow problem consists of lattice elements that express the result for each point in the program. The solution consists of a valid result for the start and the end of each node in the flow graph We refer to these results as the *input*- and the *output*-solution of a flow graph node. Given a unique number n for each node, we abbreviate the input and the output solution by I_n and O_n respectively.

The following subsections explain fundamental properties of each element of the model in more detail.

Flow Graph

The flow graph comprises the control flow structure of the program either on the inter- or intraprocedural level. Even though there are differences between these kinds of flow-graphs [1] three general graph properties are of interest for all data flow analysis: branches, join points and backward edges.

An output solution describes the situation at the end of a node. It influences the input solution of a successor node. If the node ends with a conditional branch or `switch`-instruction then there are several successor nodes. Consequently, a single output solution can contribute to several input solutions.

Join points are flow graph nodes with several predecessors. The input solution approximates the solution from several input paths by which the node can be reached. This requires the safe approximation of the output solution of *all* predecessors. Thus, several output solutions contribute to a single input solution.

[1] Intraprocedural graphs are usually *reducible* and exhibit an inherently linear structure. This reduces the complexity of solution process and allows for specialised approaches like interval-analysis.

Finally, the iterative algorithm traverses the graph in a specific order and propagates data flow facts along the edges of the graph. The order of the graph traversal defines backward edges which are edges that target a node which has already been processed. Information flowing along backward edges may influence information computed previously. An optimistic input solution may be replaced by a more conservative approximation. Such a weaker input solution may in turn yield a weaker output solution. Therefore, the classical algorithm has to iterate over nodes that it already had handled before. Section 3.1.3 will consider the influence of backward edges on the iterative algorithm in more depth.

In the intraprocedural case the nodes of the flow graph G correspond to basic blocks of the program. A basic block is a maximum sequence of instructions that can be entered at the first of them and exited only from the last of them. A basic block begins at the entry of the method, at the target of a branch, or at the instruction after a branch. *Intraprocedural* analysis considers method calls usually not as branches to keep the analysis local to the method. As a consequence, the execution of the callee is treated like an ordinary instructions.

Edges connect the nodes a and b in a flow graph if control may flow from the end of a to the beginning of b, e.g. if the last instruction of a is a branch that targets the first instruction of b. However, the flow graph model can also capture other levels of control flow abstractions as well. For example, the call graph of the program describes the calling relation between the methods of the program which can be represented by additional interprocedural edges in extended variants of the flow graph graph.

The basic block model itself is already an abstraction of a more fine-grained model, where each flow graph node corresponds to a single instruction. A basic block summarises the effects of a linear sequence of instruction nodes. This avoids the repetitive application of instruction-level transfer functions because the start state is mapped to the end state immediately. However, the instruction-level model has the advantage that only one transfer function has to be defined for each instruction to specify a data flow problem. The functional approach combines the different levels of abstraction because it computes transfer functions for larger contexts like basic blocks or whole methods from the instruction-level transfer functions automatically.

Lattice

The lattice L is a mathematical structure that defines a partial order and a greatest lower bound operator $meet : L \times L \rightarrow L$. The *meet*-operator maps two given elements to the greatest element that is smaller than the operands. In contrast to a complete order, a partial order does not require that all elements are in the order relation - i.e. there can be incomparable elements none of which is smaller or equal than the other. The most prominent example of a partial order is the power set which uses set inclusion $\subseteq$ as order relation. The sets $X = \{a, b\}$ and $Y = \{a, c\}$ are not comparable because neither X is a subset of Y

nor Y is a subset of X. However, the *meet*-operator is defined for all elements and maps X and Y to the set $\{a\}$ which is the greatest set that is a subset of X and Y.

The partial order of the lattice models the expressiveness or the quality of data flow facts. The greater the solution the stronger the assertions about the program state that have been found by the analysis. For example, if the analysis tries to find available arithmetic expressions then a set which contains more available expressions makes stronger assertions about the program state.

We say that a data flow value is *more conservative* or *weaker* than another if it ensures strictly fewer facts about the program state. The partial order usually defines a *most optimistic value* and a *most pessimistic value*. Any other value is more conservative than the most optimistic value and the most pessimistic value is more conservative than all other values. Sometimes, the most optimistic element is artificial. For example, the most optimistic value of constant propagation states that a variable may hold "any desired constant" which is not a natural assertion about any reasonable program state. Nevertheless, the extremal elements are useful for a unified handling of different data flow problems.

The meet-operator of the lattice models the *safe approximation* of data flow facts at join points. Mathematically, the meet-operator computes the greatest element which is smaller than the operands. In terms of data flow facts, the meet-operator computes the strongest assertion about the program state which is more conservative than two given facts. This definition has two implications: Firstly, the result can only ensure at most fewer facts about the program state. Thus, it *safely* approximates the input facts. Secondly, the strongest assertion which satisfies this condition is chosen. This takes into account, that the analysis tries to derive the strongest result.

The safe approximation operator models the semantics of join points in the flow graph where different execution paths meet - for example after a conditional or after a loop. If different data flow facts have been computed on the incoming paths, then only those facts which are valid on all paths remain valid after the join point. This is why the result of the meet-operation has to be at least as conservative as the operands. Secondly, no information shall be dropped without any reason. This is why the meet-operation yields the strongest element that safely approximates the operands.

Interestingly, the lattice model is sufficient to model a large number of data flow analyses in a uniform way because only the order relation and the meet-operator is required for the definition and solution of the analysis problems. For example, bit-vector analysis like reaching definitions and available expression are usually defined based on set operations which can be interpreted as operations on the power-set lattice. In contrast, different kinds of constant propagation or type inference analysis cannot be expressed in the bit-vector model but they fit smoothly into the lattice model.

Similarly, several analyses prefer to use the dual operator *join* which selects the smallest upper bound of two elements because this is a more natural represen-

tation in the set-based model. Consequently, intermediate results are refined *in lattice order* and not against it. However, this behaviour also models safe approximation. In both cases, the direction of the binary operator determines the transition from two optimistic solutions to the best - but usually more pessimistic - solution that comprises the given solutions. Without loss of generality, we use the symbol of the meet-operator $\sqcap$ to model safe approximation throughout the thesis.

Function Space

The functions of the function space T model how elements of the program like instructions or whole basic blocks influence the data flow information. They map input information to the corresponding output information. Thus, they map lattice elements back into the lattice.

Mathematically, transfer functions have to be monotone with respect to the lattice order. Thus,

$$a \sqsubseteq b \qquad \Rightarrow \qquad f(a) \sqsubseteq f(b)$$

In other words, transfer functions *preserve* the lattice order. Consequently, a conservative approximation of the input always yields a conservative approximation of the output. This property and the properties of the safe approximation guarantee the termination of the iterative solution process whenever the lattice has finite height [2] .

Label Function

The label function $[\![]\!]$ formalises the relationship between flow graph nodes and their transfer functions. It maps flow graph nodes to transfer functions. The aim is to separate a specific instance of a data flow problem - i.e. a concrete program - from the set of transfer functions. For example, analyses on Java Bytecode can be described by a single transfer function for each *kind* of Bytecode instruction. The label function maps the bytecodes of a program to their corresponding transfer function.

The formalism models the relationship between program and transfer functions efficiently. For example, different instructions may have the same effect on the data flow information. Consequently, the label function can map all these instructions to the same implementation of a transfer function. Furthermore, the composition of elementary transfer functions models the transfer function of sequential structures like the instruction sequence of basic blocks easily.

[2] Even if the lattice itself is not finite, the analysis may terminate fast, if the transfer functions reach fix points after a finite number of applications

3.1.2 The Flow Graph Model and Equation Systems

The discussion in the previous section abstracts from specific properties of a concrete data flow problem. The graph-based model is convenient to formulate graph-based solution algorithms. However, the structure of a data flow problem can also be expressed in terms of a system of data-flow equations. The benefit of equation systems is that they emphasise the dependencies between data flow values. This simplifies the formulation of the general validation principles in Chapter 4. Furthermore, the algebraic description of a data flow problem is the foundation of the summary functions model which is introduced in Chapter 5.

Essentially, transfer functions, solution elements, and the lattice with its operators have directed matches in the equation system. In contrast, the flow graph structure is encoded implicitly in the structure of data equations.

The following sections describe the mapping in detail.

Lattice Elements The equation system contains one defining equation for each input- and output-solution. We will use the abbreviations I_i and O_i as variables in this equation system. Thus, a valid solution is a mapping from data flow variables to concrete lattice elements which solves the equation system.

Transfer Functions The semantics of transfer functions can be modelled directly by an equation. Obviously, the application of a transfer function computes the output solution of a flow graph node from its input solution. Thus,

$$O_i = t_i(I_i) \qquad \forall i \in \text{FlowNodes}$$

The term t_i can be considered as a functional that selects the correct transfer function for flow graph node i form the function space T. This functional corresponds to the label function of the classical model. The notion t_i is easier to read and emphasises the relationship to the corresponding flow graph node, its input-, and its output-solution.

Any output solution O_i which is part of a data flow solution, has to satisfy the corresponding equation. If so, the solution is correct with respect to the local semantics of the flow graph node.

Flow Graph Structure and Conservative Approximation The second set of equations captures the structure of the flow graph: Each input solution has to be a conservative approximation of the output solution of all of its predecessor nodes. Consequently,

$$I_i = \bigsqcap_{j \in \text{pred}(i)} O_j \qquad \forall i \in \text{FlowNodes}$$

The fact that the safe approximation involves all predecessor nodes implicitly encodes the edges of the flow graph. Therefore, the second kind of equations capture the semantics of the control flow of the program.

Thus, the system of equations completely defines a data flow problem. It involves only lattice elements, the conservative approximation operator of the lattice and monotone transfer functions. The process of solving a data flow problem is equivalent to finding a solution of the equation system. Furthermore, the equation system provides a checking criterion for a given solution as well: a given solution is a valid solution if it solves the system of data flow equations.

3.1.3 The Iterative Worklist Algorithm

The goal of the data flow analysis is to determine a input- and output-solution I_i and O_i for each node i in the flow graph.

The essential idea of the iterative worklist algorithm is to start with an optimistic assumption for each input- and output-solution and to subsequently reduce the assumptions according to the equations which define the analysis problem until a valid solution has been found for the whole equation system.

Therefore, the algorithm maintains a worklist of nodes whose input solution has been modified because a modification of an input solution requires that the data flow information needs to be propagated. The whole algorithm can be summarised as follows:

1. Choose an optimistic initial guess I_i for each $i \in \mathit{FlowGraphNodes}$. Place every node i into the *worklist*.
2. While the *worklist* is not empty
   a) Remove node i from the worklist
   b) Compute $O_i^\star = t_i(I_i)$
   c) For all successors j of i
      i. Compute $I_j^\star = I_j \sqcap O_i^\star$
      ii. Put node j into the worklist if $I_j^\star \sqsubset I_j$

The algorithm applies transfer functions of flow graph nodes to input solutions. This way the analysis computes the effects of the execution of a program fragment given that the assertions represented by the input solutions hold. This yields new assertions O_i' about the program state after the execution of the code fragment in the flow graph node. The input solutions of all successor nodes have to be a safe approximation of these assertions which is why the data flow information has to be merged into the input solution by the conservative approximation operator $\sqcap$. This ensures that the input solution corresponds to the conservative approximation of the output solution of all predecessor nodes when the algorithm stabilises.

The initialisation and the termination of the algorithm requires some additional discussion.

Initial Guess It is important to observe that the algorithm starts with an *optimistic overapproximation* of the solution for each internal node. An obvious choice is the most optimistic element $\top$ of the lattice which states that virtually any assertion about the program state is true. The intuition is that the algorithm subsequently reduces the strength of the assertions according to the restrictions imposed by the semantics of the instructions and the control flow of the program. Therefore, the algorithm starts form the "best" of all possible solutions because this way it is possible to reduce the result to the strongest assertions which solve the data flow problem.

As a consequence, the algorithm maintains an optimistic overapproximation of the result throughout the solution process and stops when this approximation solves the data flow problem. The stepwise reduction of an overapproximation guarantees that the algorithm computes the *maximum* fix point solution of the program, because it weakens the solution only as far as necessary.

Special care has to be taken with respect to the initial solution of the entry node of the flow graph. The entry node has no predecessor. Therefore, its input solution will never be weakened. Thus, it has to be correct right from the start and must *not* be an optimistic overapproximation.

The most pessimistic element $\bot$ of the data flow lattice is a natural choice for the input solution of the start node because it represents the empty set of assumptions about the program state. However, the input solution of the entry node sometimes incorporates problem specific knowledge in order to improve the precision of the analysis. The most prominent example is the reaching definitions analysis. This analysis computes which definitions of a variable are available at each point in the program. The most pessimistic element of this analysis is the set which contains all definitions because it is safe to assume that there exists some execution path in the program by which a definition reaches the program point. Thus, the analysis strives to rule out as much definitions as possible because this strengthens the assertions about a variable at a program point.

This analysis does *not* choose the most pessimistic element as the input solution for the entry node, because this corresponds to the assumption that all definitions are available at the start of a method. This assumptions is safe but too conservative because this particular analysis can safely assume that no definition reaches the entry node of the method.

Iteration and Termination The termination of the algorithm depends on two observations: Firstly, the conservative approximation operator $\sqcap$ can only weaken the input solutions of flow graph nodes. Secondly, all transfer functions preserve the lattice order - i.e. if an input solution which is weaker than another then it can only be mapped to a weaker or equal output solution.

Observe that this does not imply that output solutions are weaker than input solutions, because the transfer functions can very well add assertions which strengthens the result. For example, the transfer functions of the reaching

definitions problem can introduce new definitions if these are generated in the flow graph nodes.

Together the monotony of the transfer functions and the monotony of the conservative approximation operator ensure that all input- and output-solutions are replaced by strictly weaker solutions only.

The algorithm iterates whenever an input solution of an already processed flow graph node is weakened because this implies that its output solution may have to be weakened, too.

The monotony property gives rise to several termination arguments for this iterative process. Firstly, if all descending chains [3] in the underlying lattice are finite, then the maximum length of a chain limits the number of iterations. The reason is that the conservative approximation operator strictly weakens the result of each input solution so that the sequence of input solutions forms a descending chain.

The algorithm can still terminate fast even if the chains in the lattice are not finite due to special properties of the transfer functions which define the data flow problem. The transfer functions of several data flow problems guarantee that a repetitive application of the functions reach a fix point after a constant number of steps. This also limits the number of iterations, due to the monotony of the conservative approximation operator and the fact that transfer functions preserve the lattice order.

3.1.4 Elimination Methods

Elimination Methods [RP86] take a slightly different approach to solve the equation system which defines a data flow problem. The general idea is that the system of equations can be solved by techniques which are closely related to Gaussian elimination.

Each equation defines a data flow variable in terms of other data flow variables. A variable in the defining term can be replaced by its own defining term. Given that at least the defining term of the entry node is a constant value this substitution strategy can solve the system of equations.

As long as the flow graph of the program is acyclic the strategy succeeds because the defining terms of the predecessor nodes can substituted into the defining terms of the successor nodes. A challenge arises at backward edges because they correspond to a self reference in the defining equation of a data flow value.

Such a cyclic dependency has to be resolved. One possible approach is to find some problem specific “loop-breaking terms” i.e. some criterion to derive a valid solution for the recurrent definition based on knowledge about the analysis. For example, it is possible to drop self-recurrent terms of a specific nesting depth for bit-vector analysis without loss of precision. The reason is

[3] A descending chain is a sequence of lattice elements where each subsequent element is smaller than the previous one

that these kinds of analysis exhibit fast convergence in a sense that they reach a fix-point immediately after one iteration of a loop has been considered.

The generic approach to resolve a cyclic dependency is to apply an iterative fix-point iteration to the self dependent term.

Thus, the elimination approach does not always solve the determination of a fix-point solution of a recursive equation, but it effectively isolates and extracts recursive equations by subsequent substitution of data flow variables. This principle is the foundation of interval-based analyses like the original Allen-Cooke interval analysis [AC76], and its improvements [HU75], [GW76], [Tar81]. Essentially, these analyses subsequently compress non-cyclic regions in the flow graph, resolve the cyclic dependency and proceed until the program has been collapsed to a single node. The analyses work well for reducible flow graphs where each cyclic sub-graph is single entry only.

The most important principle of the elimination approach is that variable substitution can be used to compress sub-regions of the flow graph. This principle gives rise to our definition of function composition in Section 5.1.

A simple example for the application of this principle is the use of transfer functions for a whole basic block instead of a transfer function for each single instruction. Assume that a basic blocks contains three instructions $i_1, \ldots i_3$. The corresponding instruction-level flow graph contains a sequence of three flow graph nodes which leads to the following equation system

$$\begin{aligned} O_1 &= t_1(I_1) \\ I_2 &= O_1 \\ O_2 &= t_2(I_2) \\ I_3 &= O_2 \\ O_3 &= t_3(I_3) \end{aligned}$$

where t_1, t_2, t_3 denote the instruction-level transfer functions. A repeated substitution of the defining terms of the variables in the last equation reduces the last equation to

$$O_3 = t_3(t_2(t_1(I_1)))$$

Thus, all intermediate states within the basic block have been removed and the final equation defines the mapping from the input solution I_1 of the basic block to the output solution O_3 of the basic block. Given that function composition is defined on the elementary transfer functions, the equation can be further compressed to

$$O_3 = t_{bb}(I_1) \qquad \text{with} \qquad t_{bb} = t_3 \circ t_2 \circ t_1$$

which *constructs* the basic block transfer function t_{bb}.

3.1.5 Advanced Scenarios of Program Analysis

The simple framework of data flow analysis has been extended to cope with several additional challenges that arise from more sophisticated application scenarios. We just provide a very brief overview here and postpone a deeper discussion to the sections where the validation of data flow results deals with some of the additional aspects.

Partial Program Analysis The traditional formulation of a data flow analysis problem assumes that the whole program is available during the analysis phase. However, software systems are usually composed of several components which interact with each other and which may even be implemented in different languages. Even the simplest monolithic programs interact at least with the operating system in a non trivial way.

Thus, data flow analysis has to deal with method invocations which target code that is not available during analysis. The first solution is to treat such interface methods pessimistically. Essentially, all information about the program state which may be influenced by the external call is dropped. This leads to analysis results which are safe but usually less precise than necessary.

Therefore, partial program analysis strives to determine analysis results in a way so that they can be composed with the results of the other components later on. This preserves the precision of the analysis but introduces additional challenges with respect to the representation of the analysis results. This aspect will be considered further in Section 4.3.

Demand Driven Analysis The original purpose of program analysis was to enable program optimisations. Thus, data flow analysis is usually not performed for its own sake but with an intended usage in mind. Therefore, it is a natural idea to reformulate the data flow problem to the question whether some interesting program property of the program holds at a specific program point. The intuition is that this question is substantially easier to answer than to determine strong assertions for each point in the program.

This led to the definition of demand driven analysis. Such analyses start from a specific analysis goal and consider relevant parts of the program only. A prominent example is program slicing [Wei81] which starts from a specific slicing criterion like the value of a single variable at a program point and restricts the program to only those parts which influence the value of the variable. To do so, the slicing approach determines the transitive closure of the data- and control-dependencies which contribute to the slicing criterion.

We discuss this general idea and its influence on the validation scenario in more depth in Section 4.3 which describes the safe lower bound principle and in Section 6.3 which applies the principle for the optimisation of the validation process.

Conditional Analysis The most important reason for the efficiency of data flow analyses is that it combines information about different execution paths very early at the join points in the program. Whenever data flow information is propagated to a successor node, then this information is merged with the previous information about the input solution before the analysis reconsiders the node. This strategy merges the information about two different paths which may contribute different kinds of assertions about the program state and continues to compute information which hold always independently from the path by which the node is reached.

This strategy effectively limits the number of paths which are actually considered by the analysis and it guarantees the fast termination of the data flow analysis algorithm.

However, it looses precision if some of the transfer functions which define the data flow problem are monotone but do not distribute over the safe approximation operator, i.e.:

$$t_i(a \sqcap b) \sqsubset t_i(a) \sqcap t_i(b)$$

The point is, that the analysis could have derived strictly stronger assertions about the program state after node i if it had followed two paths which exhibit different input solutions a and b separately instead of merging the path early by the conservative approximation $a \sqcap b$.

In order to avoid this loss of precision some extensions of the traditional data flow framework encode data flow information which hold under some conditions only. For example, the analysis may track two kinds of information depending on the value of a conditional. This technique separates the different execution paths for an if- and an else-branch for example. The more sophisticated the various kinds of analysis information get the more the analysis degenerates to an analysis which considers *all* execution paths of the program and merges different information about a program point only as a last step to construct the final solution.

This solution is called the *meet over all paths solution* and it is the best solution which can be derived by a static inspection of the program. However, this approach requires exponential effort because all potential execution paths have to be considered. In contrast, the traditional algorithm computes the *maximum fix point solution* which terminates fast but has the potential to loose precision due to the early merge of paths at join points in the program.

Interestingly, the computation of the meet over all paths solution is more closely related to the model-checking approach which is discussed in Section 3.2.

3.2 Model Checking and Abstract Interpretations

Schmidt and Steffen observed that program analysis can be considered as model-checking of abstract interpretations [SS98], [Sch98], [Ste91]. We briefly review

the key observations which justify this general statement in order to discuss how the validation of data flow results relates to the model checking approach for proving program properties.

3.2.1 Model Checking and the Relationship to Program Analysis

A model checker is a procedure that decides whether a given structure M is a model of a logical formula ϕ, i.e. whether M satisfies ϕ [MOSS99]. M is an abstract model of the program in question, which is a usually finite automata-like structure and ϕ is some kind of temporal logic that specifies the desired property.

The nodes of the model M represent abstract program states. Two program states are connected by an edge if the program state can change from the state represented by the source node to the state represented by the target node. The graph is annotated either with atomic propositions which describe the properties of a node (Kripke structures) or with actions that characterise the transitions represented by the edges (labelled transition systems).

Formulas of the temporal logic ϕ are constructed from the atomic propositions, boolean connectives, and quantifiers which range over paths in abstract model M. Thus, the formulas can express properties like "for all execution paths in M property a holds" or "there exists an execution path where property b holds" etc.

Generic decision procedures can check whether a given model M satisfies a given formula.

This very generic idea offers a variety of different modelling decisions: How are concrete program states abstracted? What are the atomic propositions? What kinds of formulas are used?

Interestingly, the model checking approach can also express data flow problems quite naturally. The general idea is to use the modelling techniques of abstract interpretation [CC77] to encode a data flow problem in terms of a model suitable for model checking. Abstract interpretation expresses the relationship between an execution of the concrete program and traces in an abstract program model. This is achieved by two mapping functions, an abstraction function α and a concretisation function γ which map executions of the concrete program to an abstract interpretation in the program model and back again. These functions form a galois connection which means that a set of concrete program states is mapped to an abstract representative and an abstract state can be mapped back to the set of concrete states it represents.

A simple way to relate concrete executions of the program to a abstract interpretation is to map any concrete state to an abstract state which merely contains the value of the program counter. The resulting abstraction is the flow graph of the program because there is one abstract state per program point and the abstract interpretation follows the flow of control.

Similarly, the semantics of the actions of the concrete program can be abstracted as well. Essentially, the concrete semantics of a single instruction that specifies how the instruction modifies the concrete state has to be mapped to a abstract semantic of the instruction which specifies how the instruction modifies the abstract state. This abstraction step is closely related to the definition of transfer functions in the traditional formulation of data flow problem. Transfer functions also specify the transformation of the abstract state, which is expressed as an element of the data flow lattice.

These observations provide a way to specify a data flow problem in terms of a model-checking problem [SS98]:

- The abstract program states in the model M correspond to the program points. Thus, the model M is a representation for the flow graph of the program.
- The actions (in a labelled transition system) represent the abstract semantics of the instructions. For example, the semantics of the program instructions with respect to the reaching definition problem reduces to actions which state that program variables are used or modified by the instruction.
- A formula in temporal logic specifies the data flow result. For example, the reaching definitions problem can be stated by a formula that states that a definition is available if there is a path from the definition of the variable which does not contain a modification of the same variable.

Obviously, a model checker can determine whether a data flow fact holds or not by checking if the model satisfies the formula. Additionally, a model checker which solves the *global model-checking problem* "Given a finite model structure M and a formula ϕ, determine the set of states in M which satisfy ϕ" [MOSS99] solves the complete data flow problem, because the problem statement translates to "Given a flow graph and a formula which describes when a definition d reaches a program point in the flow graph, determine all program points in the flow graph which are reached by a definition d". Thus, it is possible to solve a data flow problem with model-checking techniques. This observation raises the question whether model-checking techniques can also be applied for the validation of program analysis results.

3.2.2 Validation of Program Analysis Results

The program abstraction which is guided by the abstract interpretation approach yields a model checking problem which closely resembles the data flow problem. It uses the general model checking approach in a very restricted way. Therefore, the model does not suffer from the usual state explosion problems a model checker is usually confronted with. Furthermore, the model checking algorithm which determines fix points for arbitrary logical formulae seems to degenerate to the data flow algorithm which computes a fix point for the data flow problem. The observation which justifies this assumption is that both the

temporal formula and the definition of the data flow problem quantifies over the paths in the abstract representation of the program. For example, a definition reaches a program point if *there exists* a path in the *flow graph* which connects the definition with the program point on which the variable is not redefined.

This thesis assumes an application scenario in which the code consumer lacks the computational capabilities to perform the fix point computations which solve to data flow problem on its own. Thus, a model checking process which resembles the data flow algorithm will likely be too costly to be performed by the consumer. However, the inherent relationship between data flow analysis and a model-checker raises another question: is it possible to apply validation techniques for data flow problem to a model checking problem, too? The idea is that special annotations may guide the model checking process so that it does not compute but just validate fix points of temporal formulae. This may very well increase the efficiency of the checking process. The close relationship between temporal formula that specify data flow problems and the traditional formulation of data flow problems can establish the required link to adopt the fix point checking techniques presented in this thesis to the model checking algorithm. This seems to be an interesting area of further research.

Another question is why we consider the model checking solution of data flow problems at all if there is a well established framework for the solution of data flow problems at hand. The traditional formulation of data flow problems *restricts* the potential abstractions of the concrete programs in several ways. Most importantly, the executions paths of the program are abstracted into the flow graph which does not take any information about the program state into account. As a consequence, the flow graph joins different execution paths as early as possible. This is the source of the efficiency of the data flow analysis, because it effectively restricts the number of paths which have to be considered. However, it is also the source of the potential loss of precision. An analysis which tracks different paths separately before combining the results as a final step computes the so-called *maximum fix-point solution*. This solution can be more precise than the solution of the traditional fix-point algorithm for non-distributive problems.

The general model checking framework provides such capabilities because it is not bound to the efficient but simple program abstraction fixed in the definition of flow graphs. However, the separation of different execution paths usually leads to a significant increase of the number of abstract states. This directly challenges a potential application of this technique in the application scenario of this thesis because the resources at the consumer side are supposed to be limited.

Furthermore, the additional degree of freedom imposes an implementation challenge: a model checker running at the consumer side has to support different kinds of program abstractions. In contrast, the traditional formulation of data flow problems always use the same abstraction namely the flow graph of the program.

Nonetheless, the aforementioned idea of validating a fix point of temporal formulae hints at an interesting question: If it is possible to apply validation techniques to the generic model checking process itself, can this increase the efficiency in such a way that more expressive abstractions of the program executions can still be validated with limited resources? The answer to this question is beyond the scope of this thesis but as outlined previously the inherent relationship between data flow analysis and model checking may very well provide a starting point for the adoption of data flow validation techniques to a more sophisticated model checking scenario.

4 Fundamental Validation Principles

This chapter focuses on general principles which give rise to the validation of data flow results for program modules and elaborates on the key challenges for the analysis model. Chapter 5 presents a summary function model which solves these challenges and Chapter 6 describes optimisation strategies for the validation process based on the terminology established in this chapter.

The validation principles are closely related to the goals of the thesis. Firstly, the *general validation principle* states that the validation of data flow results corresponds to the check that the results solve the system of data flow equations which describe the problem in question. This principle is the core reason for the inherent efficiency of the validation process because the *validation* of a given solution requires a single pass over the system of equations only. In contrast, the *computation* of the result - which corresponds to the analysis phase - requires iteration over the system of equations.

Secondly, the *intentional under-approximation principle* shows that the validation process can validate *any* solution of the data flow problem. The principle provides an interesting degree of freedom for the analysis phase because it is possible to weaken a result as long as it stays expressive enough to achieve the goals at the consumer side. This is useful because the validation of weaker results usually reduces the effort which has to be spend during the validation process.

Thirdly, the *interprocedural validation principle* refines the general validation principle to support the validation of interprocedural results. We use the functional approach to interprocedural analysis [SP81] which consists of two phases: The first phase computes a summary function for each method in the program. Such a summary function comprises the effects of method invocation. Thus, it maps the program state immediately before the execution of the method directly to the program state upon return from the call. The second phase computes a safe approximation for the invocation context of each method. This phase takes the summary functions of the first phase into account. After that, it is trivial to derive intermediate program states for each instruction in a method from the invocation contexts and the summary function model. One of the key features of the functional approach is that the computation of summary functions is again formulated as a data flow problem. Therefore, it is possible to adopt the general validation principle to the validation of interprocedural results.

Finally, the validation of modular results introduces additional challenges. First of all, the validator has to be able to validate the modular representation of results. Fortunately, the general strategy for the validation of summary

functions can be adopted to validate their modular counter-parts as well. Secondly, the incremental scenario requires that the validator is able to safely approximate the potential effects of other software modules. This is important to ensure that the validator can safely use pieces of the result ahead of time. The same mechanism is also required to deal with the partial validation scenario as well. We express the potential effects of other software modules in terms of variables in our model. The *safe lower bound principle* states that it is always possible to replace all of these variables with a safe lower bound. This operation yields a safe under-approximation of the modular result. At the same time, the variables can act as insertion point for more precise results which in turn enables the validator to compose modular results subsequently.

The chapter is structured as follows: Section 4.1 explains the relationship between the flow graph, the data flow equation system and the transmitted certificates for an intraprocedural analysis. Subsequently, the general validation principle and the intentional under-approximation principle are considered in this simple setting. The following section starts with a review of the functional approach to interprocedural analysis, explains its different phases, and explains why the computation of summary functions is also a data flow problem. The adoption of the general validation principle leads to the interprocedural validation principle. Furthermore, the section addresses two additional challenges of interprocedural analysis: parameter passing and dynamic method binding at call sites. Section 4.3 explains the fundamental idea for the representation of partial analysis results. Furthermore, we discuss how the modular result model is used in the incremental and partial validation scenario. Finally, Section 4.4 summarises the key challenges for the analysis model which arise from the discussions in the chapter. These challenges are solved by the summary function model presented in Chapter 5.

4.1 Intraprocedural Validation

This section formulates the general validation principle and the intentional under-approximation principle in the intraprocedural setting. Both principles are stated in terms of a generic data flow problem. Thus, any data flow analysis result can be validated according to these principles. Section 4.2 applies the principles to interprocedural analysis.

Validation is primarily concerned with the data flow solution and its inherent structure. For further details about the iterative solution process which computes the data flow solutions and a general discussion of classical data flow problems refer to Section 3.1.

4.1.1 The General Validation Principle

The analysis phase performs data flow analysis for a program and transmits the program and a *certificate* which holds a data flow solution to the validator. The

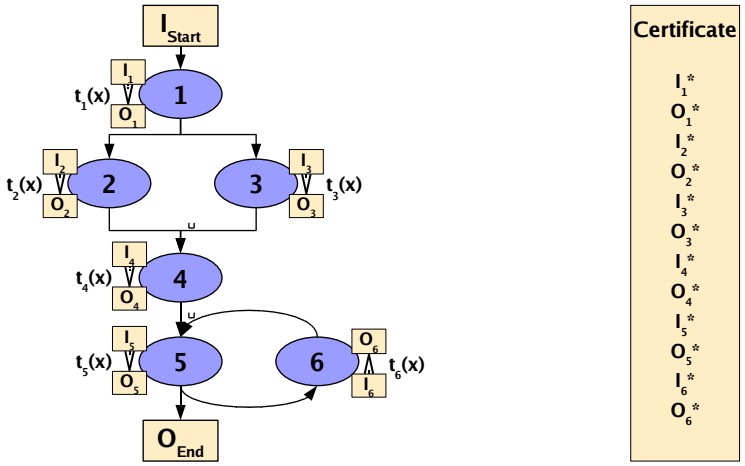


Figure 4.1: Program Flow Graphs and Certificates

term certificate is borrowed from the proof-carrying code principle where the certificate contains proofs about program properties. The notion emphasises that the certificate is precomputed during the analysis phase and holds data flow facts that describe properties of the program. The ultimate goal of the validator is to show that the data flow solution encoded in the certificate is a *valid* solution with respect to the given program. This is vital because the certificate may have been modified during transition and the use of erroneous data flow results might break the integrity of the consumer.

A natural representation of the program is a *flow graph G*. Figure 4.1 depicts a flow graph of a program and the corresponding certificate. Nodes of the flow graph model fragments of the program in question while its edges model the fact that control may flow from the end of the source node to the first instruction of the target node. As discussed in Section 3.1 the nodes of a flow graph in intraprocedural analyses represent basic blocks and its edges model the branching structure within the method. The extensions which capture interprocedural control flow are discussed in Section 4.2.

A *data flow solution* consists of an *input* and an *output solution* for each flow graph node. Input and output solutions model facts which always hold whenever execution reaches the start and the end of a flow graph node respectively.

It is important to observe that there are several instances of an input or output solution. There are input and output solutions computed during the analysis phase, input and output solutions transmitted in the certificate, and input and output solutions computed during the validation phase. We mark the input and output solutions in the certificate with an additional asterisk whenever we want to explicitly separate them from other input and output solutions.

Furthermore, we mark intermediate solutions which are computed during the validation phase with a star.

According to Section 3.1.2 the following system of data flow equations defines the fix point solution of the data flow problem $D = \langle G, [\![]\!], L, T \rangle$:

$$\begin{aligned} \forall i \in \mathit{FlowNodes}, t_i = [\![i]\!] \in T : & \\ O_i &\sqsubseteq t_i(I_i) \\ I_i &\sqsubseteq \begin{cases} I_{\mathit{Start}} & \text{if} \quad i = 1 \\ \bigsqcap_{j \in \mathrm{pred}_G(i)} O_j & \text{else} \end{cases} \end{aligned}$$

This system models the dependencies between input and output solutions, which have to hold for any valid solution. There are two different kinds of dependencies. Firstly, output solutions depend on input solutions. Output solution O_i represents the program state immediately after the execution of flow node i while input solution I_i characterises the state immediately before the execution of the node. Therefore, O_i has to reflect the modifications on I_i due to the execution of the node. This is modelled by application of transfer function t_i to the input solution. Thus,

$$\forall i \in \mathrm{FlowNodes} : O_i \sqsubseteq t_i(I_i)$$

These equations define valid output solutions under the assumption that input solutions are valid.

Secondly, input solutions depend on output solutions of the predecessor blocks. If a flow graph node is reachable only by one predecessor, then its input solution is equal to the output solution of this predecessor. If there are several predecessors then the flow graph node constitutes a join point. Different paths through the program meet each other at join points and only data flow facts which are valid on each path remain valid at the join point. Hence, the dependency of input and output solutions is defined by the *safe approximation*

$$\forall i \in \mathrm{FlowNodes} : I_i \sqsubseteq \bigsqcap_{j \in \mathrm{pred}(i)} O_j$$

These equations define valid input solutions under the assumption that output solutions are valid.

The whole equation system defines a valid data flow solution because the validity of all input solutions gives rise to the validity of all output solutions a vice versa. Thus, a given complete solution $O_1^* \ldots O_n^*, I_1^* \ldots I_n^*$ is valid if it solves the system of data flow equations. This algebraic observation can be expressed in terms of the underlying program as well: The output-input dependency enforces that the solution expresses the effects of a flow node execution correctly, while the input-output dependency enforces that the solutions capture the effects of the flow structure in the program. A solution is valid only if both dependencies hold throughout the program.

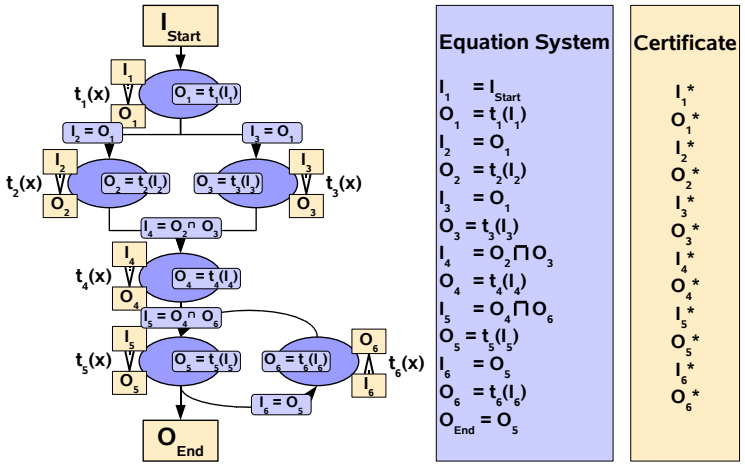


Figure 4.2: Relationship between Flow Graph and Equation System

The relationship between the flow graph model and the equation model is depicted in Figure 4.2. The definition of output solution O_i depends on the transfer function application of the corresponding node and the input solution I_i. Similarly, the safe approximation of the output solution of the predecessor nodes defines an input solution I_i. For example, node 4 has two predecessors and its defining equation $I_4 = O_2 \sqcap O_3$ models the two corresponding edges in the flow graph.

The data flow values I_{Start} and O_{End} have a special meaning. They model the program state right before the execution of the method and after its execution respectively. Thus, I_{Start} models the invocation context of the method. An intraprocedural data flow analysis uses some safe assumptions about this state as discussed in more depth in 3.1.3, which is modelled by the state I_{Start} in the equation system. Program state O_{End} summarises the program state after the method call has finished. This state is of special importance in the interprocedural context as explained in Section 4.2.

All in all the observations of this section lead to the following principle

Principle 1 (General Validation Principle) *In order to check that a data flow solution is valid the validator checks that it solves the system of inequalities that define a valid solution.*

Given that the validator receives a solution $O_1^*, \ldots, O_n^*, I_1^*, \ldots, I_n^*$ in the certificate and the code of a method, this check boils down to two different elementary steps. Firstly, the validator applies the appropriate transfer function to each given output solution O_i^* and shows that result $O_i^\star$ satisfies

$$O_i^* \sqsubseteq O_i^\star = t_i(I_i^*)$$

This ensures that the given output solution is valid provided that the given input solution is valid.

To prove this for a single input solution I_i^*, the validator computes the conservative approximation $I_i^\star$ of the given output solutions of all predecessor nodes and checks that

$$I_i^* \sqsubseteq I_i^\star = \bigsqcap_{j \in \text{pred}(i)} O_j^*$$

The validator has established the validity of the whole solution, if the checks hold for each inequality of the system.

Obviously, this check requires a single pass over the system of equations only, because each equation is evaluated once. This *single pass property* is the main reason why the validation is more efficient than the analysis phase. In order to *solve* the system of data flow equations, the solution algorithm starts with an optimistic guess and iterates over the system of equations until the solution has stabilised.

The recomputation of a data flow value is required during the iterative algorithm because the system of data flow equations usually contains recursive equation structures which origin from loops. For example, the substitution of I_6, O_5, and I_5 in the defining equation of O_6 yields

$$O_6 = t_6(t_5(O_4 \sqcap O_6))$$

Thus, O_6 depends on itself. A valid result of such a self-dependent data flow value is a *fix-point* of the corresponding recursive equation. The iterative algorithm computes such a fix-point by subsequent evaluation of the corresponding data flow equation which is why the solution algorithm may have to process equations several times.

Consequently, the validation pass can also be considered as a *fix-point test* which ensures that a given solution of a recursive equation system is valid. In contrast, the iterative solution algorithm performs a *fix-point computation* which yields a valid fix-point which is subsequently transmitted in the certificate.

4.1.2 The Intentional Under-Approximation Principle

The iterative solution algorithm computes the *maximal* fix-point solution - i.e. the solution which solves the system of data flow equations exactly. In contrast, we have relaxed the validation condition to the test that the solution solves the corresponding system of *inequalities*. This test still ensures that the solution is a fix-point but not necessarily the best one. The observation leads to the second important principle.

Principle 2 (Intentional Under-Approximation Principle) *The validator checks the validity of* any *valid solution to the data flow problem because the validation process checks the system of inequalities.*

This principle is important because it offers an additional degree of freedom to the code producer and the code consumer. The analysis phase at the producer site has enough resources at hand to run a full-fledged analysis which yields the most precise result with respect to the program. However, the analysis results serve a specific purpose: for example, to show that a specific interface is respected or to trigger some optimisations at the consumer site. Obviously, there may be pieces of the most precise result which are not necessary to show the interface compliance or which do not trigger some desired optimisations. The code producer can reduce a result to a weaker one which still serves the intended purpose before the certificate is transmitted to the code consumer. The advantage is that weaker analysis results can be represented more compact and that their validation can be significantly faster.

The validator can use the intentional under-approximation principle to protect himself against denial-of-service attacks. Whenever the validation of a single data flow fact gets to complex, the validator can choose a under-approximation which is easier to verify. As a consequence, depended values may have to be relaxed also, but the validator still proves the validity of the modified solution.

Obviously, the under-approximation principle is most valuable when the data flow results are used to trigger optimisations because the loss of optimisation opportunities does not compromise the integrity of the system. This differs from the security scenario, where the results can only be relaxed as long as they enforce the desired property.

4.2 Interprocedural Validation

Interprocedural validation is essentially a reinterpretation of the general validation principle in a more complex setting. The functional approach to interprocedural analysis operates in two phases. The first phase computes a summary function for each method which models how a given invocation context is manipulated by a invocation. Summary functions take effects of other method invocations during the execution of the method into account.

The central observation is that the computation of summary functions is itself a data flow problem. Thus, summary functions can be validated according to the general validation principle. However, the system of data flow equations describes the dependencies between *data flow functions* and not the dependency between *data flow values*. The validation of this equation system introduces special challenges for the representation of summary functions which are solved by the function model presented in Chapter 5. A combination of data flow functions and data flow values describes the result of an interprocedural analysis. We will use the term *data flow facts* whenever it is not necessary to explicitly distinguish between the different kinds of data.

Furthermore, it is important to notice that the computation of a summary function for a whole method, involves the computation of intraprocedural summaries which map the invocation context to the state at a specific program

point within the method. These intraprocedural summaries can shortcut the computation of data flow values so that the validation of interprocedural analysis results has to be reformulated in a more sophisticated *interprocedural validation principle*.

The following sections provide a short review of the functional approach to interprocedural analysis before we discuss the elementary issues of interprocedural validation in detail and establish the interprocedural validation principle.

4.2.1 Review of Interprocedural Analysis

The functional approach to interprocedural data flow analysis was originally formulated by Sharir and Pnueli [SP81]. The approach operates in two phases: the first phase computes *summary functions* which summarise how the code between two program points modifies data flow information. Subsequently, the second phase computes concrete data flow values for each program point based upon these summary functions.

The computation of summary functions itself can be separated into two subproblems: the computation of summary functions within the method and the integration of summary functions of callees.

Firstly, let's assume that the analysis is applied to a leaf method - i.e. a method which does not call any other method. The goal is to compute a summary function for each program point in the method. Such a summary function describes how the code of the method modifies the invocation context of the method and yields the solution of the corresponding program point as depicted in Figure 4.3.

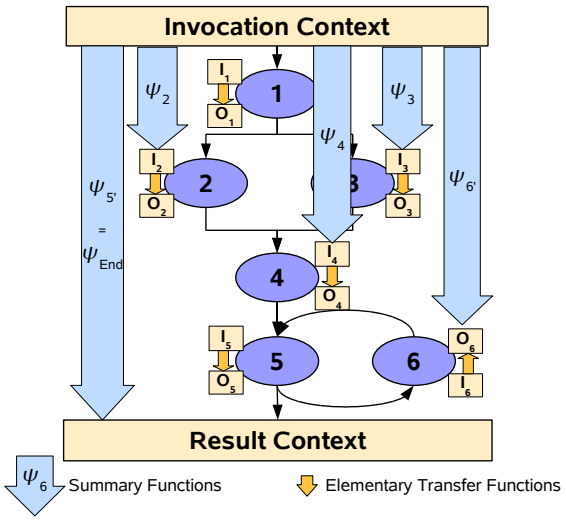


Figure 4.3: Internal Summary Functions of a Method

Let ψ_{mn} denote an intraprocedural summary function which maps the program state at point n to the program state at point m. We identify each program point with the unique number i of some flow graph node in the method. To separate the input state of a flow graph node from the output state, we mark the output state with an additional prime.

Thus, summary function ψ_{0i} maps the invocation context - which is the state immediately before the execution of the entry node 0 to the to the input state I_i. Similarly, $\psi_{0i'}$ maps the invocation context to the output state of flow graph node i. The summary functions which map the invocation context to an intermediate state play a vital role during the summary function computation and we usually omit the leading 0 in the index expression if it is clear from the context.

Each of these intraprocedural summary functions can be regarded as "shortcut" function which maps the invocation context *immediately* to the program state at a specific point in the method. Especially, the semantics of loops like the control flow from node five to node six is captured by the summary functions of subsequent states like ψ'_6. The summary function ψ_{End} which maps the invocation context to the result context of the method will be of special importance for the computation of summary functions in the presence of method invocations.

Before we explain this aspect in detail, we state the computation of summary functions as a data flow problem. The goal is to compute a summary function for each point in the method. Thus, summary functions acts data flow values of the data flow analysis which computes summary functions: instead of values of the data flow lattice each input or output solution of flow node i holds the corresponding summary function which maps the invocation context to the program state at that point. Furthermore, the transfer functions of flow graph nodes and the meet operator have to be redefined on the function level as depicted in Figure 4.4.

A transfer function of a flow graph node takes an "input" summary function - e.g. ψ_3 - of the node as parameter and yields the "output" summary function. The input summary function maps the invocation context to the program state I_3 and the *summary function of the flow graph node* maps I_3 to the output state O_3. Consequently, the output summary function $\psi_{3'}$ which maps the invocation context directly to the state O_3 can be computed by *function composition* of ψ_3 and the summary function of the flow node $\psi_{33'}$. Thus, transfer function application during the computation of summary functions reduces to function composition with flow nodes summaries.

Join points require safe approximation. Consider the flow of control from node 2 to node 4 and from node 3 to node 4 respectively. The analysis computes two summary functions $\psi_{2'}$ and $\psi_{3'}$ which map the invocation context to the program state immediately after their nodes. The assertions about the program states at these points may differ from each other so that at the start of node 4 only those assertions hold which are valid on both paths. The summary function that maps the invocation context to the state I_4 has to capture this intuition. Essentially, this involves a safe approximation operation defined *on functions*.

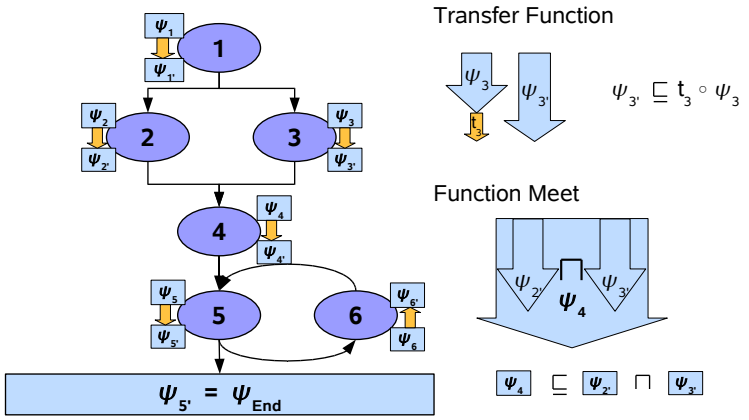


Figure 4.4: Computation of Summary Functions by Data Flow Analysis

The definition of this operation is based on the safe approximation operator $\sqcap_L$ of the inducing data flow problem. It has to hold, that

$$\psi_\alpha, \psi_\beta \in \text{SummaryFunctions}:$$
$$\psi_\gamma =_{df} \psi_\alpha \sqcap_\Psi \psi_\beta$$
$$\text{where} \quad \psi_\gamma(x) = \psi_\alpha(x) \sqcap_L \psi_\beta(x) \qquad \forall x \in \text{DataFlowValues}$$

Essentially, the meet-function ψ_γ is defined as the function which maps all parameter values to the conservative approximation of the *result values* of the given functions.

All in all the computation of intraprocedural summary functions requires the solution of a classical data flow analysis problem. The analysis just operates on a lattice of functions as and its transfer functions correspond to function composition. However, this framework is closely coupled to the inducing analysis framework: The summary functions of flow graph nodes correspond to the transfer functions of the underlying problem and the safe approximation of summary functions has to preserve the conservative approximation of the inducing value lattice.

An additional challenge arises at call sites. The goal of the summary function analysis is to compute summary functions which incorporate the summary functions of the callees of the method. To do so, the summary function of a callee replaces the transfer functions of the purely intraprocedural problem as shown in Figure 4.5.

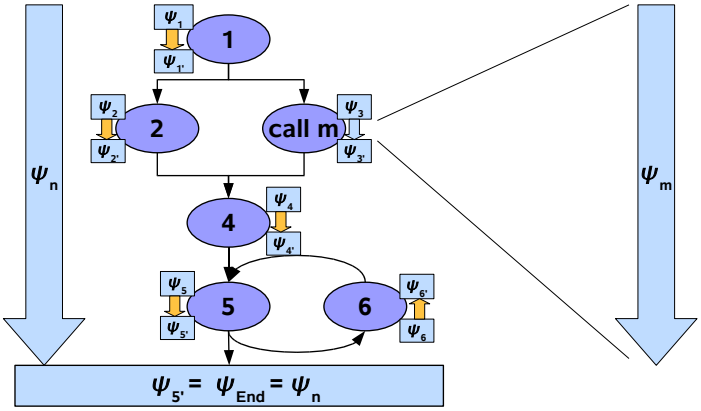


Figure 4.5: Integration of Summary Functions of Callees

The summary function of a callee corresponds to its intraprocedural summary function ψ_{End} which maps the invocation context to the program state immediately after the end node in the control flow graph. In order to emphasise the special status of this kind of function we call them *interprocedural summary functions*. In contrast, we use the term *intraprocedural summary functions* for the functions which map the invocation context to some intermediate state within the method.

For the sake of simplicity we assume that interprocedural summaries can be immediately inserted at call sites. This is only valid if all summaries manipulate a common set of global variables and if each method invocation is bound statically. For a discussion how the model is extended to deal with local variables and dynamically dispatched method invocations refer to Section 4.2.4.

The integration of interprocedural summary functions at call sites introduces additional dependencies between summary functions because the interprocedural summary of a method depends on the interprocedural summaries of callees. This dependency can even be cyclic if the method invocations are directly or indirectly recursive. The interprocedural analysis resolves these interprocedural dependencies by an extended fix-point computation. Initially, a optimistic - but usually invalid - interprocedural summary function is assumed for each method. The intraprocedural computation of summary functions uses the optimistic guesses for the computation of a new solution for the method currently under consideration. This usually yields a more conservative interprocedural summary function at the exit node of the method. Consequently, the computations of the summary functions of all callers of the actual method have to be repeated. The whole system eventually stabilises after several iterations.

Finally, the first phase of interprocedural analysis yields an interprocedural summary function for each method and intraprocedural summaries which map the calling context of a method to each intermediate state within the method. The second analysis phase uses these functions to compute data flow values which describe the program state at each program point. Interestingly, it is only necessary to compute the invocation context of each method, because all intermediates states within the method can be directly computed by the intraprocedural summaries.

The final invocation context of a method characterises the assertions about the program state which hold for *any* invocation of the method. Formally, it is the safe approximation of all invocation contexts at all call sites. These invocation contexts are intermediate results within the caller. As such they depend on the invocation context of the caller as explained above. Accordingly, we observe another - potentially cyclic - dependency in the final step of interprocedural analysis. Once again, fix point iteration resolves these dependencies.

The intraprocedural summary function for the input state of a call instruction fastens this process, because it maps the invocation context of the caller directly to the invocation context of the callee at the call site. This shortcuts the propagation of data flow values from the invocation context of the caller to the invocation context at the call sites and avoids intraprocedural fix-point computations.

Summary The review of the functional approach to interprocedural analysis fleshes out properties which are essential for the adoption of the general validation principle to the interprocedural realm:

Firstly, interprocedural analysis computes summary functions to capture the effects of method invocations. Thus, the validation of interprocedural analysis results is not only concerned with validation of data flow values but also with the validation of summary functions.

Secondly, the computation of summary functions is again a data flow problem which operates on functions instead of values. A transfer function of this problem is the composition of flow node summary function which stems from the inducing data flow problem. Only at call sites the analysis inserts interprocedural summary functions. The safe approximation operator which captures the semantics of join points is redefined in the functional setting. These observations allow the adoption of the general validation principle in the interprocedural setting.

Thirdly, only data flow values which represent the safe approximation of the invocation context of each method are of interest. Any intermediate state within the method can be derived directly from its corresponding intraprocedural summary function.

Finally, interprocedural analysis solves three different kinds of fix-point computations. Intraprocedural summary functions incorporate cyclic dependencies which arise from loops in the control flow of a method. The computation

of precise interprocedural summaries resolves recursive calling dependencies between methods and the computation of precise invocation contexts solves interprocedural dependencies of data flow values. This is important, because it shows that interprocedural analysis is inherently more complex than the inducing intraprocedural counterpart. Furthermore, the central advantage of validation is that it replaces costly fix-point iterations by a fix-point test. Therefore, the code consumer can benefit even more in the *interprocedural* setting, because he can avoid three different fix-point computations. This is even more important setting because the calling dependencies between the methods of a program are usually not that uniformly structured than intraprocedural control flow.

4.2.2 Validation of Summary Functions

The result of an interprocedural analysis consists of a summary functions for every single program point in a method and a safe approximation of the calling context of each method. Thus, the validation of interprocedural analysis results requires both, the validation of summary functions and the validation of data flow values.

The validation of summary functions can be reduced to the general validation principle. The discussion in Section 4.2.1 shows that the computation of summary functions is a variant of the generic formulation of a data flow problem which uses function composition with transfer functions and a safe approximation operator that is defined on summary functions instead and not on data flow values.

The system of data flow equations that defines a valid data flow solution (see Section 4.1.1) can be rewritten accordingly:

$$\begin{array}{rcl} \psi_{i'} & \sqsubseteq & f_i(\psi_i) \qquad \text{with} \begin{cases} \text{for} i \notin \mathit{Call}: & f_i(x) = t_i \circ x \\ \text{for} i \in \mathit{Call}: & f_i(x) = \psi_{\mathit{call}_i} \circ x \end{cases} \\ \psi_i & \sqsubseteq & \bigsqcap_{j \in \mathit{pred}(i)} \psi_{j'} \end{array}$$

where ψ_{call_i} denotes the summary function of the method called in flow node i.

Obviously, the validation that given summary functions establish a valid solution of this equation system requires only a single pass over the equation system like in the intraprocedural setting. However, the equations deal with data flow functions and not with data flow values. Therefore, the check requires a function representation which supports the following operations:

Function composition $\circ$ is required to compute a guess for an output summary function with respect to a given input summary function.

Conservative approximation of functions $\sqcap_\Psi$ is required to compute a guess for an input summary functions with respect to given output summary functions of predecessor nodes.

Function comparison $\sqsubseteq_\Psi$ is required to compare the solution guesses to the functions given in the certificate in both situations.

Especially function comparison - which is obviously crucial for the validation - requires special attention: and explicit representation of a summary function as an explicit map from input to output values is very inefficient and thus usually not a practical approach. On the other hand, more compact representations like $\psi(x) = \bot$ also allow equivalent representation like $\psi(x) = \bot \sqcap x$ which cannot be compared to each other immediately. Chapter 5 presents a generic representation for summary functions which supports all of the required operations.

Once again a special issue arises at call sites. All transfer functions t_i of the data flow problem in question are part of the trusted computing base of the validator and can be trusted. However, the summary function of a callee ψ_{call_i} is part of the transmitted result and *cannot* be trusted immediately. Therefore, the validator has to perform an additional check: the function ψ_{End_i} that is derived during the computation of intraprocedural analysis has to be at least as optimistic than its alter ego ψ_{call_i} that is inserted into the summary functions of all callers. Thus, summary functions are only valid if the additional inequality

$$\psi_{Exit_n} \sqsubseteq \psi_{call_n}$$

holds for all summary functions.

Parameter passing and dynamic method binding complicate the situation at a call site even further because a direct function composition of a single callee summary does not capture the effects correctly. However, the model can be extended accordingly to deal with this issue as outlined in Section 4.2.4.

All in all the validation of summary functions can be reduced to the general validation principle and the system of equations that describes a valid solution reveals that function composition, function meet and function comparison have to be supported by the summary function model in a *validatable* way.

4.2.3 Validation of Data Flow Values

The core challenges of the validation of interprocedural analysis results are already solved during the validation of summary functions. Checking invocation contexts is straight-forward given that *validated* intraprocedural summary functions are at hand. By definition, the transmitted calling context has to be a safe approximation of all invocation contexts of the method at all call sites of the program. The program state of a call site i in method m can be directly computed by applying the summary function ψ_i of the call site to the invocation context IC_m of the caller.

Thus, the following inequality has to hold for each given invocation context IC_n:

$$IC_n \sqsubseteq \bigsqcap_{i \in CallSites(n)} \psi_i(IC_m)$$

All other data flow values do not even have to be transmitted because they can be directly computed from the invocation context and corresponding intraprocedural summary function.

At this point it is important to observe, that the validity of a data flow solution is a global criteria. In general, *all* equations have to be checked before the validity of any data flow value can be taken for granted. However, this is a severe restriction compared to the intraprocedural scenario which establishes the validity of - admittedly less precise - data flow values after the inspection of a single method already. The underlying reason is that several data flow facts depend on intraprocedural data flow only. The conservative assumption about the invocation context and about summary functions of callees treat dependencies on interprocedural data flow safely but accept the corresponding loss of precision. This observation leads to the central idea how to detect valid lower bounds for an interprocedural analysis result at any point in time which is discussed in Section 4.3.

4.2.4 Method Invocation Semantics

The discussion of the general approach to the validation of interprocedural analysis results assumed that an interprocedural summary function can be directly used as the transfer function of a method invocation instruction. However, virtual method binding, parameter passing, and local variables complicate the issue.

Dynamic Method Binding Virtual method binding can be represented quite directly by an extension of the control flow graph of a method. The node of each call instruction is split into several nodes one for each possible call target of the invocation as depicted in Figure 4.6

The target of a dynamically bound method call depends on the runtime type of the receiver reference[1]. Therefore, a dynamic method invocation can be considered to be a `switch`-instruction which evaluates the runtime type of the receiver and branches to an invocation of a concrete implementation m_i. The control flow immediately merges again in the successor node of the original call instruction. This intuitive graph model translates directly into the equation model: the summary function of a dynamic call is modelled by the safe approximation of the interprocedural summary functions of all potential call targets. Thus,

$$\psi_{call_m} = \bigsqcap_{i \in target(m)} \psi_{call_i}$$

[1] We argue in terms of dynamic method binding in object-oriented languages because we intend to analyse Java programs. Exactly the same phenomenon arises from function variables in languages like C where the actual call target depends on the runtime value of a function pointer.

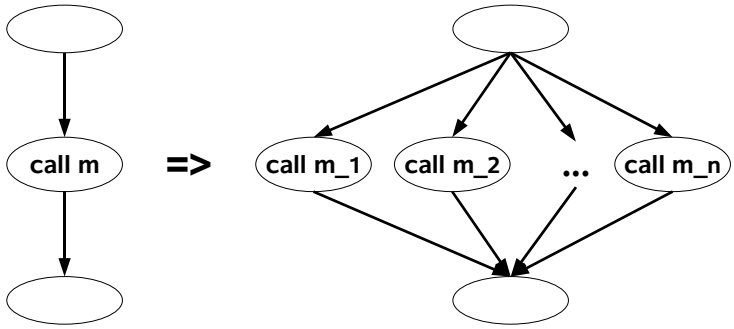


Figure 4.6: Representation of Dynamic Method Binding

This modelling idea is straight-forward but it raises an important question: how can we restrict the set $target(m)$ of "potential call targets" as much as possible to prevent that the safe approximation of the summary function looses too much precision?

A very simple approach is to take the name of the method and its signature into account. This information is available in Java bytecode at each call site immediately. However, this "name-based resolution" (NBR) approach also combines completely unrelated methods that just accidentally share the same name and parameter types. Some simple approaches improve the strategy for object oriented programs: class hierarchy analysis (CHA) takes the class hierarchy into account and rapid type analysis (RTA) restricts CHA to classes which are instantiated at least once in the program.

These approaches are efficient but not very well suited for the analysis of program modules. The reason is that all of them have to expect that some additional implementation of a method is defined in some unknown subclass of the program module. Therefore, they have to treat dynamically bound call sites conservatively until the last part of the program is available.

More sophisticated analyses perform data flow analysis to restrict the potential values of the receiver reference at a call site. This is much more convenient in our application scenario because such analyses restrict the origin of a receiver type to concrete instantiation sites. This yields a precise type which can be independent from further extensions of the program. Therefore, we have specified a simple variant of such a type inference algorithm in terms of the generic interprocedural model as discussed in Section 7.3.

The results of this analysis can be validated within our model. However, an additional issue arises: The type inference analysis restricts the receiver types

of the potential call targets and more restrictive call targets yield a more precise result of the type analysis. Thus, there exists a cyclic dependency between data flow based type analysis and the determination of call targets. This dependency can be resolved by interleaving the summary function computation and the value computation until both computations stabilise. For an exhaustive discussion of approaches to call graph construction refer to [Gro98]. Section 7.3 discusses the implications for the validation process.

Parameter Passing and Local Variables Data flow analysis computes a data flow value for each point in the program. This data flow value comprises all possible program states at the specific point for any execution of the program. Summary functions map such a program state representation from the point immediately before the execution of the code to the program state immediately after the piece of code. This comprises the effects of the code on the program state and provides a short-cut to derive the output value immediately from the input value. Thus, summary functions can be considered to be program state transformers.

It is quite natural to represent the program state by an environment which maps a set of data flow variables to data flow values. Many data flow analyses choose a one-to-one relationship between the variables of the program and the data flow variables in the environment, because they consider the flow of data through local and global variables. In this model special issues arise at call sites, because the caller and the callee operate on a separate set of local variables and the initialisation of the parameter of the callee depends on the arguments at a specific call site.

Several extensions of the original functional approach cope with this issue [Kno99], [RHS95]. We adopt the call site model of Knoop [Kno99] within our summary function representation. The central modelling idea is to express the semantics of a method call by additional "call"- and "return"-functions denoted by ψ_{call} and ψ_{ret} respectively. The call-function models the parameter passing and assigns the arguments at the call site to parameters within the callee. The return-function serves two different purposes because it maps relevant changes in the program state - like the assignment of the result value - to the appropriate place in the context of the caller and it restores the rest of the context of the caller. The whole situation is depicted in Figure 4.7.

The caller supplies arguments at the call site which determine the values of the parameters in the invocation context of the callee. Therefore, we need an additional mechanism to capture the semantics of parameter passing. Any kind of variable - local variables, parameters of the caller, and global variables - can be used as arguments of the call like the local variable l_1 which determines the value of the first parameter of m while the global variable g_1 determines the value of the second parameter. The additional summary function ψ_{call} models this mapping.

The interprocedural summary ψ_m maps the invocation context of method m denoted by IC_m to the program state immediately after the execution of the

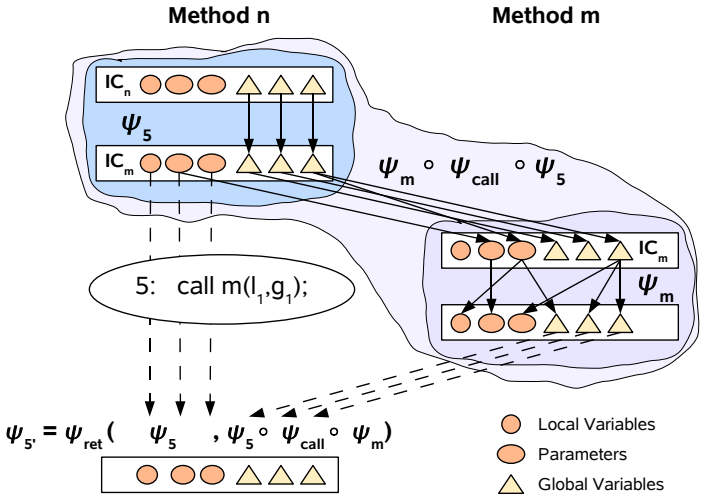


Figure 4.7: Method Invocation Model

callee. In turn, the invocation context IC_m can be derived from the invocation context of the caller denoted by IC_n by composition of ψ_5 and the appropriate call function ψ_{call}. Similarly, state immediately after execution of method m is derived from the invocation context of the caller by $\psi_m \circ \psi_{call_m} \circ \psi_5$.

However, the caller and the callee operate on separate incarnations of the method frame. Each of them has its own set of local variables and the manipulation of a local variable by the callee must not influence local variables of the caller. This is especially important for recursive method calls where caller and callee are two different invocations of the same method. In such a case the local variables of the caller and the callee have to be kept apart because the modifications of the local variables in the callee must not affect the same variables in the caller.

The composition $\psi_m \circ \psi_{call_m} \circ \psi_5$ expresses the final state of the callee *in terms of the context of the callee*. The construction of the corresponding state O_5 in the caller requires two different tasks. Firstly, modifications of the program state in the callee which affect the state of the caller - like manipulations of global variables and the assignment of the result value - have to be mapped into the context of the caller. Secondly, the values of all local variables and parameters in the caller have to be restored. This invalidates potential manipulations of the local variables in the callee. These tasks are achieved by the return functional ψ_{ret_m} which takes the input summary ψ_5 and the compositional function $\psi_m \circ \psi_{call_m} \circ \psi_5$ as parameters. The functional acts as a kind of "selector"-function which either retrieves the definition of result values form the input function in order to restore values or from the compositional function in order to integrate manipulations of the program state in the caller context. Refer to Section 5.5 for the details.

4.2.5 The Interprocedural Validation Principle

The validation of interprocedural analysis results is primarily concerned with the validation of so called summary functions. Such summary functions capture the effects of a method call more precisely than the conservative assumptions of an intraprocedural analysis. The computation of summary functions is a data flow problem which operates on data flow functions instead of data flow values. It is closely related to the inducing data flow problem, because the lattice of the inducing problem specifies the domain of summary functions and elementary transfer functions integrate the semantics of flow nodes into summary function. Thus, the general validation principle can be applied to the validation of summary functions but requires a function model which supplies function composition, function meet, and function comparison operations.

Dynamic method binding, parameter passing, and local variables complicate the model of a method invocation. This finally leads to the following validation principle:

Principle 3 (The Interprocedural Validation Principle) *The code consumer receives intraprocedural summary functions $\psi_i, \psi_{i'}$ for each flow graph node within each method, interprocedural summary functions ψ_m for each method, and a conservative approximation of the invocation context IC_m of each method.*

The check that the given values constitute a solution to the following system of equation ensures the validity of the result with respect to the program in question:

$$\psi_{i'} \sqsubseteq f_i(\psi_i) \qquad with \begin{cases} for\, i \notin Call: & f_i(x) = \psi_{ii'} \circ x \\ for\, i \in Call: & f_i(x) = \psi_{call_n} \circ x \end{cases}$$

$$\psi_i \sqsubseteq \bigsqcap_{j \in pred(i)} \psi_{j'}$$

$$\psi_m \sqsubseteq \psi_{Exit_m}$$

$$IC_n \sqsubseteq \bigsqcap_{mi \in CallSites(n)} \psi_{mi}(IC_m)$$

The construction of the summary ψ_{call_m} requires the determination of all potential call targets $target(m)$ if the call is bound dynamically.

$$\psi_{call_m} = \bigsqcap_{i \in target(m)} \psi_{call_{mi}}$$

Each single call function is constructed from a summary function which expresses the simultaneous assignment of arguments to parameters and a return functional, which restores the local context of the caller and maps the modifications due to the method invocation back into the caller context.

All in all, the general validation principle can be adopted to support the validation of summary functions and final data flow results of an interprocedural analysis. The essential difference is that the equation system deals with summary functions which have to be validated like data flow values.

4.3 Program Modules and Sophisticated Validation Scenarios

Any program is confronted with interfaces to other modules. Most state of the art programming environments like Java or C# provide a rich set of basic functionality in a runtime library. Furthermore, large software systems have to be separated into modules to keep them maintainable and to enable reuse. However, even a monolithic program written in a specific programming language interacts with the operating system by calls to system routines which provide low-level IO or access to the file system. Thus, any practical approach to program analysis has to consider the boundary between software modules written by different code producers and potentially implemented in a different way.

There are three different approaches to the analysis of software modules [CC02]:

Worst-Case Assumptions: The analysis of a software module makes conservative assumptions about the potential effects of each external call. This corresponds to the loss of all analysis information at such a call site and leads to a significant loss of precision.

User-Defined Interfaces: The analysis uses external information about the behaviour and potential influence of external calls. This information may be supplied by the user or it may stem from a separate analysis of the other software module. However, the other module can contain call-backs into the using module which have to be treated conservatively. This also results in a loss of precision.

Symbolic-Relational Analysis: Each software component is considered in isolation but the analysis yields a result which captures the dependency on other software modules. A subsequent composition phase can combine the analysis information for different modules and resolve the dependencies. This yields a precise result for the whole program but requires additional analysis effort in the combination phase.

The benefit of modular analysis is twofold. Firstly, it separates the analysis effort. The analysis of a single module can be performed in isolation and the analysis results can already be used to optimise the module. Secondly, the analysis results of a single module can be reused several times if the module is used in different contexts.

The symbolic-relational approach can even achieve the same precision of global analysis. At the same time the relational representation - if compact and

efficient - provides a natural source of speed-up. All internal dependencies within the module can be resolved during the analysis of the module so that the subsequent analysis which composes the results of different modules and resolves the remaining inter-module dependencies is significantly faster than a whole program analysis which starts from scratch.

Furthermore, a symbolic approach also subsumes the other approaches: If it is possible to integrate the potential analysis results for other modules into modular results of a software module, then we can also use this mechanism to integrate worst-case assumptions of user defined analysis results instead of analysis results. Thus, a symbolic modular result representation can significantly increase the flexibility of an analysis framework.

The advantages and the need of modular analysis naturally leads to the question how a validator can check results which stem from modular analysis. Firstly, we reconsider the equation system to find out how analysis results of other software modules can influence the result. This leads to the *safe lower bound principle* which captures the idea that assumptions about external modules can be integrated at insertion points in a symbolic representation. After a brief example which provides an intuition of the idea, we discuss how the safe lower bound principle is applied in the incremental and partial validation scenario.

4.3.1 The Safe Lower Bound Principle

Our is to find a representation which supports both the integration and validation of interprocedural data flow information and the early extraction of analysis results which do not depend on the other methods. The fundamental modelling idea arises from an inspection of the data flow equation system which describes an interprocedural analysis problem.

The first observation is that there is one defining expression for each data flow fact. The invocation context of a method is defined by the invocation contexts at each call site where the method may be invoked. Such an invocation context corresponds to the intermediate state within the caller immediately before the execution of the method. This state is defined by the corresponding intraprocedural summary function of the program point. This summary is itself defined by the summary functions of the predecessor blocks which either depend on elementary transfer functions or the interprocedural summaries of the callees. All in all the data flow facts - values as well as functions - transitively depend on each other.

What are the unknown parts within this equation system if only a single method is considered?

Obviously, the invocation context of the method is unknown, because it depends on the invocation contexts at each call site. Thus, an invocation context in an external module can weaken the result. Secondly, the summary functions of external callees depend on the behaviour of the code in these methods.

An unknown invocation context of a method acts as the parameter for intraprocedural summary functions. The evaluation of the intraprocedural summary function yields the final result of the intermediate state at the corresponding program point. Thus, the intermediate states of the method indirectly depend on the invocation context.

The intraprocedural summary functions of a method under consideration depend on the summary functions of callees because the unknown callee summaries are integrated by function composition at call sides.

Therefore, it is not possible to solve the system of data flow equations because it incorporates data flow variables and function variables which refer to *external entities*. These variables describe how the result depends on external modules. Now assume that it is possible to modify the equation system in a way, which removes all *internal variables* from the defining term in each equation so that only external variables remain. Then this result representation expresses symbolically how the data flow result of the software module depends on external code.

The summary function model developed in Chapter 5 supports the computation of such a representation. At this point we just formulate the safe lower bound principle, under the assumption that a modular result representation exists which contains variables for unknown invocation contexts and callee summaries of external methods.

Principle 4 (Safe Lower Bound Principle) *The substitution of all external variables in the system of data flow equations by safe lower bounds yields a safe lower bound for the solution of the equation system.*

The safe lower bound principle captures the observation that it is possible to construct a safe solution from a modular result representation if we replace the dependencies on external modules by pessimistic assumptions.

Now, we just briefly discuss an illustrative example and defer the definition of the underlying model to 5. Consider the simple program in Figure 4.8 and assume that the analysis in question performs copy constant propagation.

Both intermediate program states O_1 and O_2 depend on the invocation context of the method. The state O_2 additionally depends on the interprocedural callee summary ψ_m.

The following data flow equations summarise the situation:

$$\begin{aligned} O_1 &= \psi_{1'}(IC) \\ O_2 &= \psi_{2'}(IC) \\ \psi_{1'} &= t_1 \circ id \\ \psi_{2'} &= \psi_m \circ \psi_{1'} \end{aligned}$$

Obviously, we cannot compute the final values for O_1 and O_2 until IC is available and we cannot compute the summary function $\psi_{2'}$ either because the summary

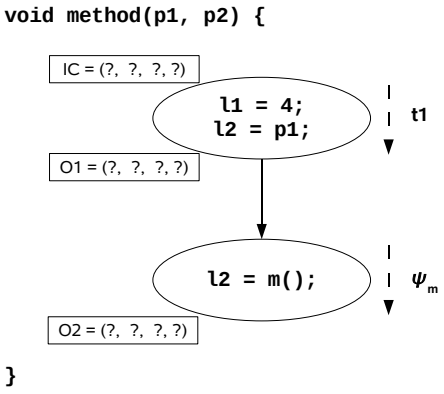


Figure 4.8: Safe Lower Bound Principle

also depends on the external callee summary ψ_m. However, it is possible to determine $\psi_{1'}$ because the function just depends on the transfer function t_1 which is given by the specification of the analysis problem.

Furthermore, we can apply the safe lower bound principle, if we substitute the unknown value of IC by the safe lower bound $IC^{\perp} = (l_1 = \perp, l_2 = \perp, p_1 = \perp, p_2 = \perp)$ and apply the known summary function $\psi_{1'}$ which yields a safe lower bound for O_1

$$O_1^{\perp} = (l_1 = 4, l_2 = \perp, p_1 = \perp, p_2 = \perp)$$

Interestingly, this safe lower bound contains the valuable information that l_1 is constant at point O_2. This information is stronger than the most pessimistic assumption about O_2 because some pieces of data flow information is generated *within* the software module under consideration. Thus, the application of the safe lower bound principle extracts those pieces of the result which hold already.

The principle can be applied to external callee summaries, too. In the example the effect of the call of method m can be safely approximated by a function which does not return a constant value. Thus,

$$\psi_m^{\perp}((l_1, l_2, p_1, p_2)) = (l_1, \perp, p_1, p_2)$$

This safe lower bound for the callee function can in turn be used to derive a safe lower bound for the output state O_2. It just has to be applied to the safe approximation of the invocation context like the preceding summary function $\psi_{1'}$. This operation yields:

$$O_2^{\perp} = \psi_m^{\perp}(IC^{\perp}) = (l_1 = 4, l_2 = \perp, p_1 = \perp, p_2 = \perp)$$

The safe lower bound principle extracts valuable information at point O_2 again, because the result states that the local variable l_1 is constant 4. This result is reasonable, because the safe approximation of the callee summary implicitly encodes, that the local variables of the caller are not affected by the method invocation. This is correct for the analysis under consideration. This way, even safe approximations of external summary function have the potential to propagate valuable data flow information which supports the extraction of information from a modular result representation.

4.3.2 Incremental Validation

In the incremental validation scenario we want to use the modular result representation to

- extract those pieces of the results which just depend on the properties of the program module so that the validator can use them ahead of time
- determine the valid parts of the available results in order to improve the efficiency of the validation process.

The idea is to split the analysis context into smaller pieces and to annotate each piece with a modular result representation that shows the dependencies on external code and with the final result of the analysis of the original module. As a consequence, each single piece of software can be considered in isolation and the validity of the final result can be established in an incremental way.

The modular result representation and the safe lower bound principle form the corner stones of the approach. The safe lower bound principle is able to extract a safe lower bound for the final data flow result from the modular result at any point in time. The validator can establish the validity of a single data flow fact as soon as the safe lower bound for the fact corresponds to the final result.

Reconsider the example program in Figure 4.8 and assume that the implementation of the callee m is given by

```
int m() {
  return 5;
}
```

Thus, the final output state $O_2^* = (l_1 = 4, l_2 = 5, \dots)$ supplied in the annotations states that both the value of local variable l_1 and the value of local variable l_2 is constant. However, the application of the safe lower bound principle yields $O_m^\perp = (l_1 = 4, l_2 = \perp, \dots)$. Thus, the fact that the local variable l_2 is constant cannot be derived from an inspection of the original method alone. Nevertheless, the fact that l_1 is constant 4 can already be used ahead of time.

As soon as the callee m becomes available it is possible to validate that its summary function returns the constant value 5, because method m does not depend on any callee. The valid summary can in turn be integrated into the

intraprocedural summaries of the callees. This directly yields the validity of the dependent result O_2^* in the caller.

Section 5.4 treats this incremental strategy in more depth.

4.3.3 Partial Validation

The incremental validation approach assumes that the analysis context is separated into several sub-modules. Each sub-module is annotated with a modular result representation which reveals its dependencies on other submodules. Furthermore, the annotations contain the final results from the analysis of the whole context, too.

The incremental validation can derive save lower bounds from the modular representation which are immediately usable. Furthermore, the final results can be checked in an incremental way. Essentially, a piece of the final result is valid if it does not longer depend on unavailable modules. This property can be checked by the comparison of the final result and the safe lower bound that safely approximates the effects of all missing modules.

The partial validation scenario differs because we expect that each software module is analysed in isolation. Therefore, it is not possible to ship the final result that incorporates the effects of external modules together with the module. However, it is still possible to ship the modular result representation and to apply the safe approximation principle to derive a safe lower bound for the software module. Furthermore, it is still possible to incorporate analysis results from other modules into the modular representation later. At this point, the modular result representation differs from the result for the software module under the worst-case assumption and it is possible to construct more precise results.

However, the fact that no final solution is available limits the effectiveness of the composition. The problem is that the result from different modules can cyclically depend on each other. A fix-point iteration is required *at the consumer side* to resolve such dependencies. In fact, this is again a data flow problem and we expect that the consumer is not able to solve such a problem on its own - even the problem is less complex than the original one, because all dependencies within each module have already been resolved. The only way out is to use safe-under approximations whenever the composition would lead to a cyclic dependency.

In contrast, the final result which is shipped in the incremental scenario constitutes the "inter-module" fix-point of the analysis, so that the validation of this precise result gets possible.

Obviously, there is a correlation between the size of the analysis context and the remaining effort during the composition in the partial analysis scenario. However, we do not consider this quite advanced scenario in detail and restrict the implementation to the incremental scenario which already deals with the

most important issues of the representation and validation of modular analysis results.

4.4 Summary and Comparison

This chapter formulates the central principles for the validation of data flow results for incomplete programs. The general validation principle states that data flow results can be validated by the proof that they solve the system of data flow equations which describe the data flow problem with respect to the given program.

The idea is also applicable to the validation of interprocedural analysis results. Such results depend on summary functions which capture the effects of methods more precisely and which are computed by a data flow analysis. The corresponding equation system is more complex and involves composition, meet, and comparison operations on summary functions. Furthermore, dynamic method binding, parameter passing, and local variables complicate the integration of callee summaries at call sites.

The validator cannot establish the validity of summary functions which depend on missing program parts. However, the validator can compute safe lower bounds for the analysis results of a software module. The idea is to consider references to missing program parts as variables and to substitute these variable by safe lower bounds. Safe lower bounds provide a safe under-approximation of the analysis result at any point in time. Furthermore, given results can be considered valid as soon as they correspond to the safe lower bound. This is an indirect proof that the given results do not depend on the results of other modules anymore.

Several issues remain to be solved:

- A function representation is required that supports function composition, function meet, and function comparison (see Chapter 5).
- The validation process relies on a valid determination of all potential targets of a call. Essentially, the validity of a call graph has to be proved. However, there exists a cyclic dependency: the call graph is constructed by an interprocedural data flow analysis, which determines values for function pointers or reference types but the analysis requires a call graph (see Section 7.3).
- Parameter passing and the return from a function call have to capture the semantics of the data flow problem correctly. The problem is discussed in more depth in 5.5.
- An incremental or partial validation requires a modular result representation. This representation must also be *validatable* because otherwise the validator cannot safely use the representation to extract safe lower bounds (see Section 5.4.4).

Comparison to Other Approaches The interprocedural validation principle is formulated on a high level of abstraction. It adopts the generic formulation of Sharir and Pnueli [SP81] which state interprocedural analysis as a data flow problem which operates on the function lattice induced by an arbitrary underlying data flow problem. At this level of abstraction, the model is both flow- and context-sensitive. Context-sensitivity is implicit because summary functions map the invocation context of a method to an intermediate state. In contrast, flow-sensitivity depends on the properties of the elementary transfer functions of the inducing problem.

Parameter passing and local variables require special attention at call sites. We adopt the return-function model of Knoop [Kno99] to deal with this issue. Reps also provides a solution in terms of a graph based function representation in [RHS95].

We observe that dynamic method binding requires additional analysis effort to restrict the potential call target while the original formulation assumed a single known target at each call site. Any such call target determination has to be validated before the results of a concrete analysis can be checked. Simple call graph analyses like name based resolution, class hierarchy analysis, and rapid type analysis [TP00], cannot be checked until the whole program is available. This also applies to more sophisticated analyses like field or method type analysis, which separate types reachable by a field or an method respectively. In contrast, analysis which consider the data flow of types [Gro98] fit into the general data flow model and can be validated according to the general principles.

The validation of intraprocedural data flow results was first addressed in the special scenario of lightweight bytecode verification [RR98]. The general applicability to intraprocedural analysis problems is addressed in [Ros03] and [Amm07].

The abstraction carrying code approach [APH05] reformulates the general validation principles in terms of an abstract interpretation framework [CC77]. The underlying constraint solver provides support for the incremental validation of data flow results [AAP06] but the framework does not supply explicit support for interprocedural analysis.

SafeTSA [ADvRF01], [vR05] approaches mobile code security from a slightly different angle. SafeTSA provides a program representation which implicitly enforces the desired properties of the program - i.e. it is not possible to represent program which violates the security constraints. The approach is based on static single assignment form which is difficult to extend to the interprocedural scenario.

5 A Generic Model for Summary Functions

Chapter 4 discusses the general validation principles which supports the validation of interprocedural analysis results for software modules. Furthermore, the chapter identifies several challenges which have to be addressed by an analysis framework which supports the validation of analysis results. This chapter presents a summary function model which supplies the required properties and which supports the interprocedural validation. The summary function model "lifts" a definition of an inducing data flow problem from the instruction-level to the interprocedural-level automatically. This way, it is possible to apply the same validation process to various kinds of inducing data flow problems.

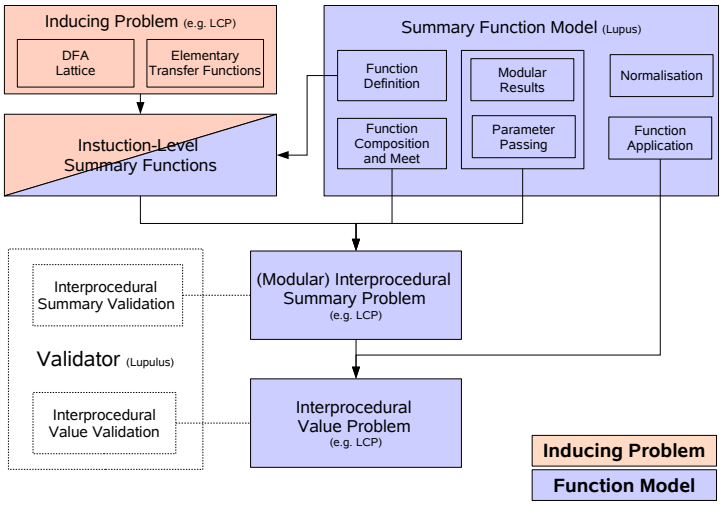


Figure 5.1: The Role of the Function Model in the Validation Scenario

Figure 5.1 shows how the separate pieces of the function model establish the bridge between an inducing data flow problem and the interprocedural validation phase. Furthermore, the figure acts as a road map for this chapter.

The inducing data flow problem has to supply the implementation of the lattice which encodes the data flow values the analysis intends to compute. This is sufficient for example to specify simple bit-vector analysis in the summary function model. Additionally, the definition of more sophisticated analysis like linear constant propagation requires the use of so called *elementary transfer functions*.

The summary function model and the elements of the inducing data flow problem are combined in *instruction-level* summary functions. The specification of an analysis requires the definition of these instruction-level summary functions only, because the function model deals with all other aspects of the interprocedural analysis problem and the validation of its solution.

The solution of an interprocedural data flow problem requires to solve two different data flow problems. Firstly, a summary function has to be computed for each method, in order to capture the effects of a method invocation at each call site in the program more precisely than in an intraprocedural analysis. Secondly, the conservative approximation of the invocation contexts at all potential call sites of a method yields a more precise result for the invocation context of this method, which in turn leads to more precise results for the data flow facts that describe the intermediate states in the method.

The system of data flow equations, which specifies the summary function problem, involves *function composition* with instruction-level summary functions, *function meet*, and an order relation on summary functions. The function model presented in this thesis supplies these operations. In particular, the model provides a simple criterion to compare two different function representations with each other. This way, the function model lifts problem specific instruction-level summary functions to the corresponding interprocedural problem automatically.

Two additional aspects complicate the computation of summary functions. Firstly, the semantics of the parameter passing mechanism has to be specified whenever the analysis deals with local variables in the program. Secondly, our application scenario requires the representation of modular analysis results. We want to supply analysis results for each software module so that the validator can continuously validate and combine the sub-solutions into a solution for the whole program. The summary function model solves these two issues, too.

The second data flow problem specifies a valid solution for the safe approximation of the invocation context of each method. This specification depends on the *application of summary functions* of the first problem. A definition of function application is also given by the function model. Summary functions have to be applicable, because they act as transfer functions which map the invocation context of a method to intermediate states within the method during the second analysis phase.

Finally, the summary function model provides a normalisation mechanism for summary functions. The normalisation is vital to keep the size of summary functions under control. This is important because a data flow solution for the first problem is expressed in terms of summary functions and has to be transmitted to and processed by the validator.

This chapter is structured according to these different aspects of the summary function model.

Section 5.1 defines the summary function model and addresses the fundamental requirements for the validation process namely how the model represents

summary functions and supports function composition, function meet and function application. The general idea is to model summary functions by *data flow expressions* which consist of elements of the inducing data flow problem and to reduce the operations on summary functions to operations on these data flow expressions. The core model of data flow expressions combines values of the inducing lattice, data flow variables which refer to the input state of the summary function and the conservative approximation operator of the inducing problem. This is already sufficient to define the instruction-level transfer functions of simple bit-vector analyses. In order to increase the expressiveness of the summary function model, Section 5.2 describes the integration of *elementary transfer functions* into the model. Such elementary transfer functions encode problem-specific properties of the inducing analysis which cannot be expressed by the core model.

Section 5.3 defines reduction rules which lead to a normal form of summary functions. The normal forms separate the summary functions into equivalence classes so that the comparison of summary functions reduces to the comparison of normal forms. Furthermore, the normal forms are compact because the normalisation process corresponds to a partial evaluation strategy of data flow expressions. Furthermore, we prove that the specified summary functions form a lattice. This ensures that they can be used to define an interprocedural data flow problem. This is the formal justification that the general validation principle is applicable for the validation of interprocedural results which are expressed in terms of the summary function model.

Section 5.4 extends the model with *function variables* in order to deal with modular results. Function variables express the dependencies on code which is external to the software module under consideration in a flexible way. It is possible to substitute such variables by safe lower bounds or to substitute them with analysis results of other software components as soon as they become available. The analysis phase as well as the validation phase can use modular analysis results in various ways. The analysis phase deals with potential effects of external code either pessimistically by a safe approximation of the function variables or optimistically if this is justified by special knowledge about the application scenario or about language properties. The validation phase can use a modular result to validate pieces of the analysis result even before the complete result has been transmitted to the code consumer. This is useful, because the validator can already use pieces of the result ahead of time and it can early drop those pieces of the result which are not required for the validation anymore.

We integrate the support of local variables and parameter passing into the model in Section 5.5. The summary function of a callee cannot be integrated directly into the summary function of a caller because both operate on their own set of local variables. The arguments at a specific call site initialise the parameters of the callee. Furthermore, the original values of the local variables of the caller have to be restored after the execution of the callee and all effects on the caller like the assignment of the result value or modifications of global variables have to be mapped into the context of the caller.

The chapter concludes with a summary and a discussion of related work which is organised according to the different modelling aspects. Additionally, we directly compare the function model with the IDE-framework of Reps, Sagiv, and Horwitz [RHS95], [SRH96] in Sections 5.1 and 5.2 because both models share several fundamental modelling ideas.

The main contribution of this thesis is that it develops a summary function model which supports the validation of interprocedural results with minimal assumptions about the inducing analysis. Furthermore the thesis shows how the model can be extended to cope with modular analysis results.

5.1 Summary Function Definition

A summary function ψ_{mn} maps the program state at point m to the state at point n and comprises the effects of all executions paths between these two points. This section defines the structure of the summary function representation and specifies the function operations which are required for the validation of interprocedural data flow results that are represented in terms of the model. At the end of the section we will show how to use the model to specify instruction-level summary functions for a specific data flow problem. Throughout the whole chapter simple data flow problems like different variants of constant propagation serve as a running examples.

The following sections extend the core model by elementary transfer functions, normalisation rules, and function variables. The fundamental modelling ideas can be summarised as follows:

1. The program state is decomposed into an *environment* - i.e. a mapping from an arbitrary set of data flow variables to data flow values. Dependencies between different pieces of the program state can be captured precisely in such a fine-grained model.
2. The representation of a summary function consists of *data flow expressions* which reduce the summary function computation to operations supplied by the inducing data flow problem. The inducing data flow problem is defined by instruction-level transfer functions and a value lattice only. Thus, the summary function model "lifts" the definition of an intraprocedural analysis to an interprocedural analysis in a generic way.
3. The summary function model supplies a simple *comparison criterion*. The existence of an efficient comparison operation is vital for the validation process.
4. We define a set of *normalisation rules* which reduce a data flow expression to a canonical form. The reduction process corresponds to a partial evaluation of the expressions and it is essential to keep the size of the function representation under control.
5. Finally, the use of *function variables* in data flow expressions can model the potential effects of unavailable parts of the program. The function

variables can either be substituted by summary functions as soon as the corresponding code becomes available or their effects can be safely approximated at any point in time. This additional degree of freedom supports an incremental validation scenario where the validator subsequently validates and integrates analysis results for classes which are loaded at different points in time.

5.1.1 Summary Functions and Data Flow Expressions

We start with a definition of the program state in terms of an environment which maps a set of arbitrary data flow variables to data flow values.

Definition 1 (Program State) *Let $Var = \{x, y, z, \dots\}$ denote an arbitrary set of data flow variables and let L be the lattice of data flow values of the inducing analysis. Then we model the program state at a program point m by an* environment *env_m, i.e. a mapping from data flow variables to data flow values:*

$$env_m = \langle x \to x_m, y \to y_m, z \to z_m, \dots \rangle$$

Thus, the variable x refers to some data flow fact "x", while x_m denotes the value of the data flow fact x at program point m.

Our central modelling idea is to define the semantics of a summary function ψ_{mn} with respect to a single data flow fact x by the following equation

$$x_n = f^x_{mn}(env_m) \qquad \text{with} \qquad f^x_{mn}(env_m) = e^x_{mn}|_{[x:=x_m, y:=y_m, \dots]}$$

The function f^x maps the program state at point m to the value of x at point n denoted by x_n. We call function f^x_{mn} *evaluation function* of x because the evaluation of the expression e^x_{mn} yields the result of the function. The data flow expression e^x_{mn} is the *defining expression* of f^x_{mn}.

Evaluation functions and their defining data flow expressions are superscribed with the name of the data flow fact they evaluate to. It is important to observe that an evaluation function takes the *whole* environment as parameter but evaluates to a *single* data flow flow value for x. A *summary function* which manipulates the whole environment consists of a tuple of evaluation functions - one for each data flow fact. Thus,

Definition 2 (Summary Function) *The summary function ψ_{mn} which maps the program state env_n at program point n to the program state env_m at point m is defined by*

$$\psi_{mn} = \langle f^x_{mn}, f^y_{mn}, f^z_{mn}, \dots \rangle = \langle e^x_{mn}, e^y_{mn}, e^z_{mn}, \dots \rangle$$

Figure 5.2 shows an example for the structure of the summary function and the environment in a small program where the program states consists of three local variables x, y, and z only. However, the model extends smoothly to greater environments as depicted on the left hand side of the figure.

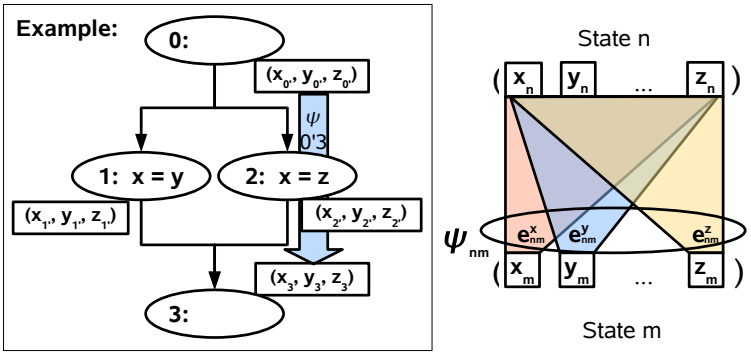


Figure 5.2: Environments and the Summary Function Model

The example program contains four basic blocks. In order to separate the program states immediately before and after the execution of a basic block, we mark the post state with an additional prime. Thus, the summary function $\psi_{0'3}$ maps the program state *after* the execution of node 0 to the program *before* the execution of node 3. Summary functions which map the state 0 of a method to some intermediate state j play an important role during the analysis phase which computes interprocedural summary functions. Thus, we omit a leading 0-index if it is clear from the context - i.e. ψ_j corresponds to ψ_{0j}.

For the sake of simplicity, we abbreviate the environment $env_m = \langle x \to x_m, y \to y_m, \dots \rangle$ by $(x_m, y_m, \dots)$ and we notate function definitions which take an environment as parameter similarly to function applications in a programming language, thus $\psi_{mn}(env_m) = \psi_{mn}(x, y, \dots)$.

The summary function $\psi_{O'3}$ consists of three evaluation functions $\langle f^x_{0'3}, \dots, f^z_{0'3} \rangle$ each of which is in turn specified by its defining data flow expression. Many of the traditional analysis choose a direct correspondence between the variables of the program and the data flow variables to model the program state. However, data flow variables in the set *Var* can also refer to different program entities like available expressions, global fields etc.

The definition of data flow expressions, which define the evaluation functions completes the summary function model.

Definition 3 (Data Flow Expression) *A* data flow expression *e has the form*

$$e \quad ::= \quad c \quad | \quad x \quad | \quad e_1 \sqcap_L e_2 \quad | \quad t_i(e_1, \dots, e_j) \quad | \quad s_i(e_1, \dots, e_{|Var|})$$

where c is a data flow value of the inducing lattice, $x \in Var$ is a data flow variable, $s_i \in FctVar$ is a free function variable, $\sqcap_L$, and $t_i \in ET$ are the safe approximation operator and an elementary transfer function of the inducing data flow problem.

This definition assumes that the inducing data flow problem is a meet-problem so that the safe approximation of two elements is given by the greatest lower bound operator $\sqcap_L$. Join-problems are treated similarly but we stick with the symbol of the meet-operation throughout the thesis. Both join and meet model the concept of *safe approximation* of two analysis results.

The different kinds of data flow expressions deal with several aspects of the data flow problem in question:

Constant Expressions (c) do not depend on the input environment. They model the *generation* of data flow facts.

Data Flow Variables (x) refer to specific elements of the input environment. They can express value assignments etc. and act as insertion points during function application and function composition (see Section 5.1.2).

Safe Approximation Expressions ($\sqcap_L$) model the safe approximation of two data flow facts in the inducing lattice L. This is vital do reduce the function meet to the meet-operator of the inducing lattice.

Elementary Transfer Functions (t_i) model more complex dependencies between data flow facts. They are required to increase the expressiveness of the model to data flow analyses like linear constant propagation.

Function Variable Expressions (s_i) act as insertion points for summary functions that model the effects of external code.

After the introduction of the summary function model, we continue with the definition of the operations on summary functions.

5.1.2 Function Operations

The definition of the required function operations is straight-forward and can be summarised as follows:

Function Application → evaluation of expressions with data flow variables substituted by parameter values

Function Composition → substitution of data flow variables with defining expressions

Function Meet → meet of expressions

Function Comparison → *structural* comparison of defining expressions

Function Application and Composition Variables in expressions give rise to the definition of function application and composition because they describe how a single item of the output state - namely x - depends on the pieces of the input environment. The evaluation function $f^x_{mn}(x, y, z, \dots) = e^x_{mn}$ can contain references to pieces of the input state like x, y, or z. A concrete input state $env_m = (x_m, y_m, z_m)$ yields the value of x at program point m by substitution of variables in e^x_{mn} with the corresponding values in env_n , thus

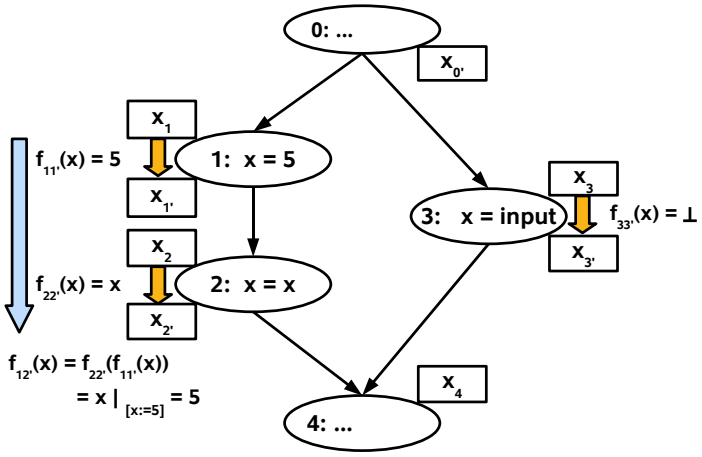


Figure 5.3: Evaluation Functions

$$\forall v \in Var, f^x_{mn}(x, y, z, \ldots) = e^x_{mn} : \qquad x_n = f^x_{mn}(x_m, y_m, z_m, \ldots) =_{def} e^x_{mn}|_{[v:=v_m]}$$

Thus, the application of the evaluation function the data flow value 7 yields $f_{22'}(7) = e^x_{22'}|_{[x:=7]} = x|_{[x:=7]} = 7$. Obviously, $e^x_{22'} = x$ models the identity function for variable x which is natural because it captures the semantics of the self assignment in the block, that does not change the value of x.

Similarly, function composition reduces to substitution of variables in expressions, too. Consider the evaluation functions $f^x_{11'}$ and $f^x_{22'}$ in Figure 5.3. The functions map the program state at point 1 and point 2 to the value $x_{1'}$ and $x_{2'}$, respectively. The evaluation function $f_{12'}$ maps the program state at point 1 to the value $x_{2'}$ directly and can be constructed as follows:

The evaluation function $f^x_{11'}$ defines the state $x_{1'} = f^x_{11'}(x_1)$ in terms of x_1 while $f^x_{22'}(x_2) = e^x_{22'}$ defines the state $x_{2'}$ in terms of x_2. Furthermore, the states $x_{1'}$ and x_2 are equal, so that $x_2 = x_{1'} = f^x_{11'}(x_1) = e^x_{11'}$. Consequently, the defining expression $e^x_{11'}$ can substitute x_2 in $e^x_{22'}$. This yields a defining expression $e^x_{12'}$ which describes the dependency of $x_{2'}$ to the input state x_1. Thus,

$$f^x_{lm}(x) = e^x_{lm} \quad , \quad f^x_{mn}(x) = e^x_{mn} : \\ f^x_{ln} = f^x_{mn} \circ f^x_{lm} =_{def} f^x_{mn}(e^x_{lm}) = e^x_{mn}|_{[x/e^x_{lm}]} = e^x_{ln}$$

Essentially, the substitution removes the variables which reference the intermediate state at the point between the two functions. For example, the substitution within the identity expression in $e^x_{22'}$ point 2 effectively propagates the defining expressions from point 1 so that the evaluation function $f^x_{12'}$ becomes the

constant expression 5. Interestingly, constant expressions like $\perp$ in $f_{33'}$ stop the propagation of expression from the preceding functions, because they do not contain variables that can be substituted. This way, newly generated data flow facts invalidate the knowledge derived for a variable beforehand.

The example assumes that the program state only consists of a single data flow fact x. However, the composition can be extended to the composition of summary functions which operate on a whole environment easily. The difference to the single variable case, is that the defining expression in the second function $f_{22'}$ can contain several data flow variables. *Each* of these variables has to be substituted with the defining expression of its corresponding evaluation function in $\psi_{11'}$.

Function Meet and the Order Relation of Functions The flow of control merges at join points in the program. After the join point only those data flow facts remain valid which are valid on all paths which reach the join point. This is captured by the safe approximation operator $\sqcap_L$ of the inducing data flow lattice because it yields the strongest data flow fact which subsumes the given facts.

We reduce the meet of summary functions to the meet of expressions. Consider the situation in the example program in Figure 5.3 where two summary functions map the input state $x_{0'}$ to the two data flow values $x_{2'}$ and $x_{3'}$ immediately before the join point denoted by the program state x_4. The meet of these two functions maps the input state $x_{0'}$ directly to the state at the join point. This state is defined by the conservative approximation of the predecessor states $x_4 = x_{2'} \sqcap_L x_{3'}$ which in turn are defined by the defining expressions of $f^x_{12'}$ and $f^x_{33'}$ respectively. Thus, the meet of these expressions captures the semantics of the join point and defines the function meet:

$$f^x_{14} = f^x_{12'} \sqcap_\psi f^x_{33'} =_{def} e^x_{12'} \sqcap_L e^x_{33'} = 5 \sqcap_L \perp$$

The definition of a meet operation always gives rise to the definition of an order relation because $x \sqcap y = y \Leftrightarrow x \sqsupseteq y$. Accordingly, the meet of data flow expressions leads to a simple criterion to decide the order relation of expressions.

Theorem 1 (Simple Order Relation on Expressions) *An expression e_1 safely approximates an expression e_2 if it contains strictly more subexpressions than e_2. Two expressions are equal if they contain exactly the same subexpressions.*

Functions are in order relation if their defining expressions are in order relation. We defined function application by expression evaluation. Furthermore, the evaluation of a meet expression can only yield a weaker result due to the semantics of the meet in the inducing lattice. Therefore, an evaluation function which combines strictly more subexpressions with this operator can only produce weaker or equal results. This way a structural comparison of data flow expressions gives rise the comparison of summary functions.

Unfortunately, the simple comparison criterion raises an important challenge. It compares two expressions purely syntactically. As a consequence, *semantically* equivalent expressions like $4 \sqcap_L 3$ and $\perp$ are *not* considered to be equal. The meet of these expressions yields

$$(4 \sqcap_L 3) \sqcap_L \perp = 4 \sqcap_L 3 \sqcap_L \perp$$

Thus, the result expressions tend to be larger than necessary. We solve this problem by the definition of normalisation rules - e.g. folding of constant expressions or the use of specific properties of the bottom element $\perp$ - which lead to a much more compact representation. This is discussed in Section 5.3.1.

5.1.3 Specification of Instruction-Level Summary Functions

The specification of a data flow problem in the functional approach to interprocedural analysis only requires the definition of transfer functions for each instruction of the program. The approach automatically combines transfer functions for two subsequent instructions by function composition. Similarly, the function meet combines the summary functions of different execution paths between two program points into a summary functions that characterises the effects of both path. This way, the functional approach computes summary functions which span larger and larger program parts.

This construction strategy by function composition and function meet does not depend on the inducing problem. However, the functional approach usually treats the function representation and the implementation of composition and meet as a black box. The summary function model presented in this Chapter goes a step further, because it does also define function composition and function meet independently from the inducing data flow problem. This reduces the specification of a data flow problem to the specification of instruction-level summary functions in terms of the summary function model and the frameworks supplies a generic implementation for function composition and meet.

For example, consider the program in Figure 5.4 and assume that we want to specify the reaching definitions problem in terms of the summary function model. There are three definitions of the local variable x which we name according to their program position as $x1, x2$, and $x3$. Furthermore, the local variable y is defined at point 0 which additionally introduces the definition $y0$. The reaching definitions analysis determines whether or not a definition of a specific variable is available at a specific program point. Thus, an environment $env_n =< x1 \rightarrow \text{bool}, x2 \rightarrow \text{bool}, x3 \rightarrow \text{bool}, y0 \rightarrow \text{bool}\rangle$ which maps a data flow variable for each definition to a boolean value can model the program state with respect to the analysis problem. The boolean value just states whether there exists an execution path in the program by which the definition can reach the corresponding program point or not.

The instruction-level transfer functions $\psi_{11'}$, $\psi_{22'}$, and $\psi_{33'}$ model the fact that the definitions $x1, x2$ and $x3$ become available at the program points. Furthermore,

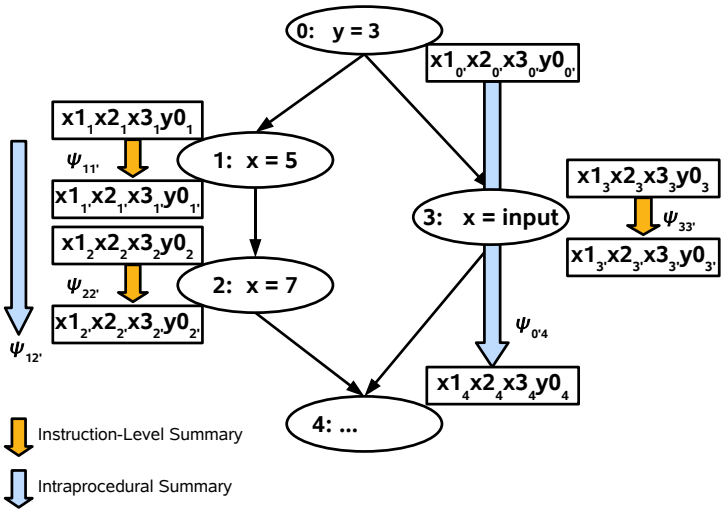


Figure 5.4: Specification of Instruction-Level Summary Functions

they invalidate the availability of the other definitions of x and preserve the availability of y is not affected. Thus,

$$\begin{aligned}
\psi_{11'} &= \langle e^{x1}_{11'}, e^{x2}_{11'}, e^{x3}_{11'}, e^{y0}_{11'} \rangle = \langle \bot, \top, \top, y0 \rangle \\
\psi_{22'} &= \langle e^{x1}_{22'}, e^{x2}_{22'}, e^{x3}_{22'}, e^{y0}_{22'} \rangle = \langle \top, \bot, \top, y0 \rangle \\
\psi_{33'} &= \langle e^{x1}_{33'}, e^{x2}_{33'}, e^{x3}_{33'}, e^{y0}_{33'} \rangle = \langle \top, \top, \bot, y0 \rangle
\end{aligned}$$

where $\bot$ (or `true`) denotes that the definition reaches the point after the instruction and $\top$ (or `false`) denotes that the definition fails to do so [1]

These instruction-level summary functions define the reaching-definition problem in terms of the summary function model. The summary function computation phase can construct summary functions which span larger contexts than a single instruction by the generic definition of function composition and function meet automatically.

For example, the function composition of $\psi_{11'}$ and $\psi_{22'}$ yields the summary function $\psi_{12'}$ by variable substitution in the defining expressions of $\psi_{22'}$:

$$\psi_{12'} = \langle \top, \bot, \top, y0 \rangle$$

[1] Traditionally, reaching definitions is modelled as a join-problem but we stick with our convention to use the symbol $\sqcap$ to denote safe approximation. The second aspect which is surprising at the first glance is that a new definition maps the definition in question to the most pessimistic element $\bot$ and all other definitions of the same variable to the most optimistic element $\top$. The reason for this is that the knowledge about the program state *increases* if *less* definitions for a specific variables have to be taken into acount. Thus, the important information gain of a new definition is that all other definitions do not reach the program point immediate after the definition in question.

This summary is equal to $\psi_{22'}$ because the single variable $y0$ is substituted by itself and constant expressions do not change during the substitution process. The function meet at the join point 4 yields the summary function

$$\psi_{0'4} = \langle \top \sqcap \top, \bot \sqcap \top, \top \sqcap \bot, y0 \sqcap y0 \rangle$$

which comprises the effects of both execution paths from the point after instruction 0 to the point immediately before instruction 4. The summary function states that definition $x1$ does not reach point 4 and that definitions $x2$ and $x3$ can reach point 4. Moreover, the fact that definitions $x2$ and $x3$ reach point 4 does not depend on any information about the program point $0'$. In contrast, the reachability information about definition $y0$ is propagated by the summary function, because the corresponding part of the mapping is essentially the identity mapping.

5.1.4 Relationship to IDFS-problems

The core model of the summary function representation presented in this section is closely related to the summary function model of Reps, Horwitz, and Sagiv [RHS95].

The design goal of the summary function model of Reps is to reduce the summary function computation to a graph reachability problem. To achieve this, a summary function is modelled as a bipartite graph in which edges connect nodes which represent the input state to nodes which represent the output state. The graph model also decomposes the program state into an environment because a node in the bipartite graph represents a single element of the whole program state.

The manipulation of the environment is modelled by graph edges in the following way. Consider the instruction-level transfer functions for the three different kinds of instructions shown in Figure 5.5.

The graph model expresses the generation of new data flow facts (new definitions) by connecting the output nodes of the definitions to an artificial `true`-element. This enables the reduction to a graph reachability problem because the question if a definition is available boils down to the question whether there is a path to the `true`-element in the graph or not. The data flow expression model avoids the additional element in the program state tuple and represents the generation of data flow facts by the constant expression $\bot$.

A new definition of a variable invalidates the reachability information about all other definitions of the variable. The graph model *implicitly* represents the invalidation of data flow facts by *missing* edges in the graph. As a consequence, there is no path to the additional *true*-element which is interpreted as the fact that definition $x2$ does not reach the point after instruction 1. The assignment of the constant expression $\top$ models the invalidation of data flow facts *explicitly* in terms of data flow expressions.

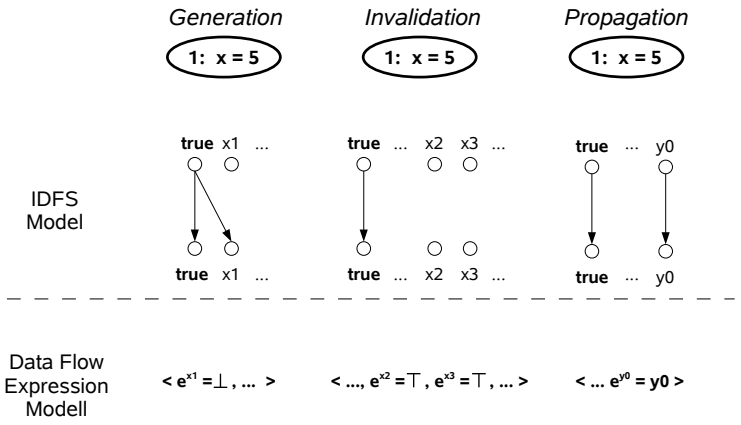


Figure 5.5: Comparison to the Summary Function Model of Reps

The propagation of data flow facts connects input nodes directly to the corresponding output node in the graph model while the expression model captures the situation by a self assignment of variables.

Function composition reduces to path compression and function meet reduces to the union of two graphs. The result graphs directly fit the corresponding data flow expressions.

Thus, the graph-based model is comparable to the expression model as long as simple bit-vector analysis like reaching definitions are considered. The differences become apparent when the approaches are extended to analyses like linear constant propagation which require more than propagation, generation, and conservative approximation of data flow facts. Refer to Section 5.2.3 for details.

5.2 Function Application Expressions and Elementary Transfer Functions

Constant expressions, data flow variables and safe approximation expressions already deal with the generation of data flow facts, assignment semantics, and the safe approximation summary functions at join points. Furthermore, the summary function model splits the program state into a tuple of data flow values to keep potential manipulations as local as possible.

These basic parts of the function model can express simple bit-vector problems and copy constant propagation directly in the summary function model as discussed in the Sections 7.1.1, 7.1.2, and 7.2.

Linear constant propagation is one of the simplest analysis which calls for an extension of the model because it cannot be specified solely with the simple types of expressions. Consider the statement

```
x = 2 * y + 10
```

Obviously, variable x is constant after the execution of the instruction if variable y is constant before. However, the relationship between the value of x and the value of y cannot be expressed by a simple assignment and it does not involve the safe approximation of different data flow facts either. The reason is that the value of x depends on the value of y in a complex problem-specific way which cannot be expressed with elements of the core model.

In order to capture such dependencies, we permit that the inducing analysis supplies a set of *elementary transfer functions*. Each of these elementary transfer functions captures a complex dependency between some data flow values in the input state of an instruction and a single value in the output environment. For example, the linear constant analysis can characterise the semantics of the statement by the linear function $x = lin_{(2,10)}(y) = 2 * y + 10$. We call the transfer functions of the inducing data flow problem elementary transfer functions to separate them from the summary functions which describe the semantics of a instruction-level summary functions in the function model.

The central idea is to use elementary transfer functions in the defining expressions of instruction-level summary function only if the manipulation of the program state cannot be expressed by simpler expressions. This way, elementary transfer functions increase the expressiveness of the summary function model, while their potential effects are kept as local as possible.

5.2.1 Properties of Function Application Expressions

Let T be the set of elementary transfer functions of the inducing data flow problem. We integrate these transfer functions into the data flow expression model as follows:

Definition 4 (Elementary Function Application Expression) *Let $t_i \in T : L^n \to L, n \in [0..|Var|]$ and $e_1, \ldots, e_n \in E$ be an elementary transfer function and data flow expressions respectively. Then the* elementary function application expression $t(e_1, \ldots, e_n)$ *is a data flow expression.*

Observe, that we allow elementary transfer functions to have an arbitrary arity n. Thus, it can take more than one data flow value from the input environment as parameter. This differs from the usual definition of transfer functions [KU77] but allows to streamline the representation of function application expressions in Section 5.4.2. The restriction to a fixed number of data flow values is vital to decrease the number of parameter expressions. For example, linear constant propagation only requires unary functions because it considers only arithmetic dependencies between a single input variable and a single output variable.

The summary function model is intended to deal with several inducing data flow analyses uniformly. Therefore, the model does not take specific properties of the elementary transfer function into account. Nevertheless, elementary transfer functions stem from the definition of a data flow problem so that they always exhibit two important properties:

1. Elementary transfer functions are monotone with respect to the order relation of the inducing lattice. Thus, they preserve the order relation when they are applied to different values.
2. Elementary transfer functions can be applied to concrete values of the inducing data flow lattice.

Furthermore, we identify an elementary transfer function $t_i \in T$ by a unique index i and we assume that the maximum set of data flow variables that can influence the result is known. The second prerequisite ensures, that we can identify the data flow variables which yield the parameters of the expressions exactly.

These properties guarantee that we can easily integrate elementary transfer functions into the summary function model. Furthermore, we can estimate their potential effects on the result of a summary function. Consider the extended example in Figure 5.6.

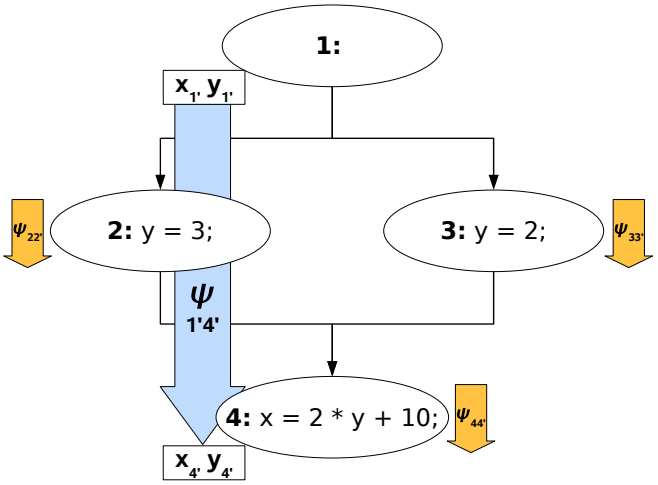


Figure 5.6: Estimating the Effects of Function Application Expressions

The summary function $\psi_{1'4'}$ results from the conservative approximation of $\psi_{1'2'}$ and $\psi_{1'3'}$ in point 4 and the subsequent composition with the instruction-level summary $\psi_{44'}$ to:

$$\psi_{1'4'} = \langle e^x_{1'4'}, e^y_{1'4'} \rangle = \langle lin_{(2,10)}(3 \sqcap_L 2), 3 \sqcap_L 2 \rangle$$

Obviously, this function always yields the most pessimistic element $\perp$ for both x and y independently from the fact whether x or y have a constant value at point $1'$. We can come to this conclusion, because both the safe approximation operator of the inducing analysis and the elementary transfer functions can be applied to values of the inducing data flow analysis. Thus, expression $e^{x}_{1'4'}$ evaluates to $lin_{(2,10)}(3 \sqcap_L 2) = lin_{(2,10)}(\perp) = \perp$. This is the most pessimistic element of the constant propagation lattice and it states that the corresponding variable is not a constant. The important observation is that it is possible to reason about the properties of elementary transfer functions even without taking problem specific knowledge about elementary transfer functions into account. This is vital for the formalisation of the normalisation process in Section 5.3.1.

5.2.2 Nesting Depth and Fix-Point Properties

The introduction of elementary transfer functions and the example in Figure 5.6 uncover an interesting property of data flow expressions. The structure of a data flow expression encode different execution paths in the program which contribute to a specific data flow value. This is closely related to the discussion in Section 3.1.2 where we observe that the flow graph of a program is encoded in the system of data flow equations which define a data flow problem. For example, the function application expression in the summary function $\psi_{1'4'}$

$$lin_{(2,10)}(3 \sqcap_L 2)$$

encodes that the two paths which join in point 4 contribute two different constants that are merged by the safe approximation operator. The result of this merge operation defines the input state for the function application expression which in turn extends the path to the point after the execution of the arithmetic operation. The evaluation of the different subexpressions corresponds to a compression of execution paths - which is one of the key ideas of the normalisation process for summary functions which is discussed in Section 5.3.1.

If data flow expressions encode data flow along different execution paths then it is a natural question, what happens in the case of cyclic structures like loops which introduce potentially infinite execution paths in the program. Consider the example in Figure 5.7.

The code contains a loop which increases the value of variable x by the factor of two which is captured by the elementary transfer function $lin_{(2,0)}$. The summary function $\psi_{1'3}$ results from the iterated composition of the summary function of node 2 namely $\psi_{22'} = \langle lin_{(2,0)}(x) \rangle$ and a subsequent conservative approximation with the previous value of the summary $\psi_{3'1'}$ to:

$$\psi_{1'3} = (1 \sqcap lin_{(2,0)}(1) \sqcap lin_{(2.0)}(1 \sqcap lin_{(2,0)}(1) \sqcap \ldots) \ldots$$

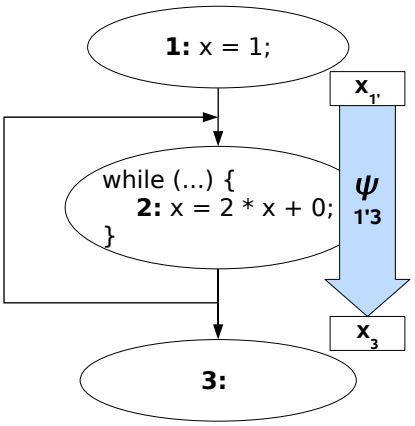


Figure 5.7: Nesting Depth of Function Application Expressions

Each of the subexpressions in the outermost conservative approximation expression corresponds to a summary function which yields the input state of one iteration of the loop. The parameter expressions of the elementary transfer function refer to the input state of the preceding iteration. Obviously, cyclic structures in the flow graph of the program can lead to an infinite nesting depth of data flow expressions, if all execution paths are stated explicitly in the data expressions.

The problem can be approached in two different ways. Firstly, the nesting depth of data flow expressions can be restricted to a constant number. Whenever a nested expression is to be substituted at this nesting level, the expression is approximated by the most pessimistic element $\perp$. This approach restricts the number of subsequent applications of elementary function applications considered by the analysis. It is safe and applicable to all inducing data flow problems but potentially looses precision.

Secondly, the summary function model can take special properties of the inducing data flow analysis into account. Any inducing data flow problem has to guarantee termination. Therefore, there can only be a limited number of elementary transfer function applications before the computation of data flow analysis reaches a fix-point. Consequently, it suffices to track a limited number of nested application expressions in the function model to ensure that the summary functions represent the evaluation sequence which computes the fix-point. For example, following the terminology of [MR90] data flow problems which are *fast* (i.e. they are 2-bounded) are guaranteed to reach a fix-point after at most two subsequent applications of a specific elementary transfer function. It is important to observe that "k-boundness" is a property of the inducing function space and not of the inducing data flow lattice. Therefore, a data flow problem can be bounded even though the underlying data flow lattice has infinite

height. The boundness of summary functions still guarantees the termination of the analysis after a finite number of iterations.

However, we will not consider the second approach further in order to keep the summary function model as lean as possible. Our primary goal is to solve all analyses presented in Chapter 7 in a more precise way than their intraprocedural counterparts. This can already be achieved with the first strategy.

Interestingly, the challenge does not even arise for simple bit-vector problems. The specification of instruction-level summary functions - e.g. for the reaching-definitions problem - does not require elementary transfer functions. Thus, it is not possible to construct arbitrarily large expressions without function application expressions. The basic expression model is limited to a conservative approximation expression which combines constant expressions and data flow variables. Thus, the size of such an expression is bounded by the number of data flow variables and the size of the inducing lattice [2].

5.2.3 Relationship to IDE-problems

The effects of nested elementary transfer functions are also addressed in the extension of the graph reachability approach to distributive environment problems [SRH96]. The extension enables the graph based model to express analysis problems like linear constant propagation.

The fundamental idea of the extension is to attach unary functions to each of the data flow edges in the bipartite graph. This way, the linear constant propagation problem can be expressed as depicted in Figure 5.8.

Each of the edges is labelled by a function which expresses the linear dependency of one constant value to another constant value. For example, the edge from a to b in the first function representation expresses the fact that b evaluates to the value $2a+1$. This linear dependency is implemented in the unary function $lin_{(2,1)}$ which maps constant values appropriately.

Function composition boils down to substitution as shown by the blue arrows in the figure. The two linear dependencies from b to a and from c to b are comprised into a new linear dependency from c to a. The IDE-model computes the resulting linear dependency $lin_{(2,5)}$ by the composition of the linear dependencies $lin_{(2,1)}$ and $lin_{(1,5)}$. This happens during the composition of the summary functions in the graph-based representation, when the edges of the input functions are relaxed to the edges of the result function.

The data flow expression model, expresses composition by substitution of expressions. However, the summary function model is *not* aware of the special semantics of linear constant dependencies. Such a dependency shows up as an expression that models the application of the elementary transfer function

[2] The folding of constant expressions during the normalisation of a data flow expression even ensures that each normal form of a data flow expression contains one constant expression only (see Section 5.3.1).

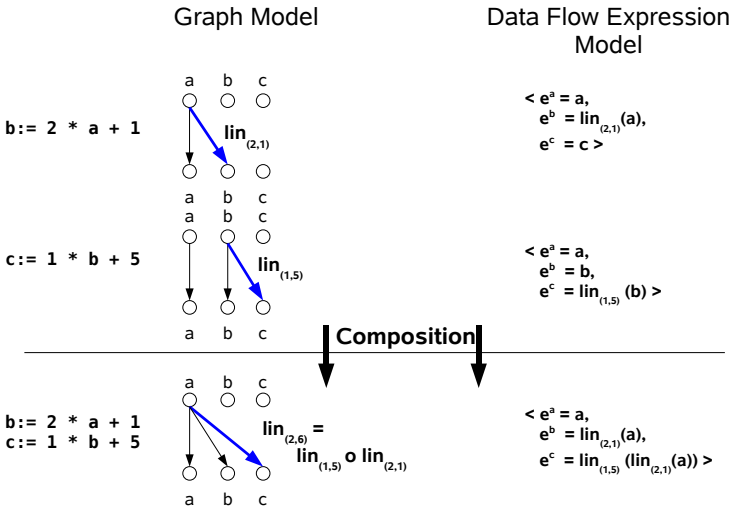


Figure 5.8: Composition of Transfer Functions of Linear Constant Propagation

application expression $lin_{(2,1)}(a)$. The substitution step during the composition of summary functions yields the *nested* expression $lin_{(5,1)}(lin_{(2,1)}(a))$ which describes the dependency between c and a.

The first observation is that the graph model of the linear constant propagation model incorporates *problem specific knowledge* into the function representation. Essentially, it is mandatory that the function composition of elementary transfer functions is computable. Furthermore, the fix-point iteration during the summary function analysis requires, that there exists an order relation on function representations.

In contrast, data flow expressions model function composition explicitly by nested elementary transfer functions. The model treats elementary transfer functions symbolically and exploits only that each function can be identified by a unique index and that it can be applied to data flow values. We restrict ourselves by such weak assumptions because we want to separate the normalisation of summary functions from problem specific implementations of elementary transfer functions. The *validation* process operates on a normal form of summary functions in order to reduce the memory requirements of the summary functions during validation. We discuss this issue in more depth in Section 5.3.

The second observation is that the expression model is not restricted to *unary* transfer functions. The model intentionally allows elementary transfer function of arbitrary arity. Such an extension is difficult in the graph model because each edge in the graph has a single source node only.

Therefore, dependencies on several input variables can only be expressed as long as they can be decomposed into the conservative approximation of dependencies on single variables. This is sufficient for linear constant propagation,

because the dependency of a single variable to each of the input variables, can be expressed by a single linear functions at each incoming graph edge.

As soon as a single function application expressions requires more than one parameter, the original graph model fails. A straight-forward extension would require the introduction of multi-edges which may significantly complicate the implementation and formal justification of the model. A well-known example of a problem which requires more than one parameter is integer constant propagation with symbolic execution of arithmetic operators because operators like addition and multiplication depend on two operands in a non-trivial way.

However, this specific analysis is not distributive and as such also beyond the scope of the current expressiveness of our function model, too. Nevertheless, distributivity is only required for the justification of the current definition of the normalisation process. It is an interesting question, whether there is a formal argument that *all* elementary transfer functions of a distributive data flow problem can be decomposed into the meet of unary elementary transfer functions or not. If it is possible to find a counter-example, then the expression model extends the expressiveness of IDE-problems even in its current formulation.

Nevertheless, the main contribution of this thesis is an investigation of the validation process and not necessarily the specification of complex data flow problems. Furthermore, we show how the support of modular analysis can be integrated directly into the summary function model.

5.3 Normalisation and Properties of Summary Functions

The definition of the summary function model in Section 5.1 raises three important challenges:

- We have to show that the summary functions can be used to encode the result of the summary function computation phase of an interprocedural analysis. If the property holds, then we can apply the general validation principle for the validation of summary functions. The proof requires to show that summary functions form a function lattice with respect to the function meet operation $\sqcap_\Psi$.
- We have to prove that the summary functions can act as transfer functions of the value computation phase. This justifies, that the validated summary functions can be used to check the result of the interprocedural invocation contexts of the methods. The proof requires to show that the application of summary functions is monotone with respect to the inducing lattice of data flow values.
- The straight forward approach to the definition of an order relation of expressions is inconvenient because it just compares expressions on a purely syntactical basis. As a consequence, several obvious transformations are

not yet exploited. For example, if a constant propagation analysis is encoded in the model, then the expressions $4 \sqcap_L 3$ and $\bot$ are *not* considered to be equal under the simple definition of the order relation of expressions. As a consequence, the meet of these expressions would yield

$$(4 \sqcap_L 3) \sqcap_E \bot = 4 \sqcap_L 3 \sqcap_L \bot$$

which is by far more complex than necessary.

This section addresses these issues and is structured in the following way: Firstly, we define normalisation rules for data flow expressions which lead to a normal form of the evaluation functions and the summary functions they define. Secondly, we consider the properties of the normal form of expressions in Section 5.3.2. Finally, Section 5.3.3 shows that the properties of normalised data flow expressions guarantee that the summary functions have the required properties.

Figure 5.9 depicts the overall line of reasoning. We start from the definition of

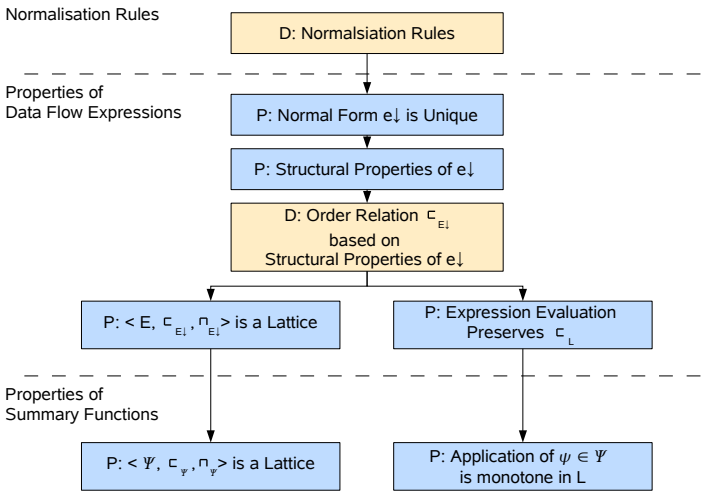


Figure 5.9: Line of Reasoning about Data Flow Expressions and Summary Functions

the normalisation rules and prove that the resulting normal form is unique. This proof can be established by showing that the normalisation rules terminate and that they are locally confluent. Next, we prove that the normal form exhibits special structural properties and define a order relation on expressions based on these properties. The order relation and the corresponding approximation operation define a lattice on the set of data flow expressions. Furthermore, the evaluation of expressions preserves the order relation of the inducing lattice - i.e. if a data flow expression is evaluated with weaker values as a substitution for variables than the evaluation does not yield stronger results. These two properties of data flow expressions give rise to the proof that the summary

functions form a lattice and that function application is monotone with respect to the order relation of the inducing data flow problem.

5.3.1 Normalisation of Data Flow Expressions

This section defines normalisation rules for data flow expressions. The normalisation serves two different purposes. Firstly, it compresses the representation of defining expressions. Secondly, normal forms of expressions can be compared to each other easily by comparing their syntactic structures. This is important to define the order relation on expressions in a generic way, which does not involve problem specific knowledge about the inducing data flow problem.

The normalisation rules can be considered as a partial evaluation of data flow expressions. Essentially, they reduce all subexpressions, which cannot depend on the input state of the summary. There are not only reduction rules that evaluation constant expressions but also rules that drop variable subexpressions that cannot contribute to the result of the whole expression anymore.

Normalisation Rules

An expression in normal form consists of the conservative approximation expression of a single constant value, data flow variables, and function application expressions where each function occurs only once and whose parameter expressions are also in normal form. The following normalisation rules lead to this normal form.

The first three rules deal with data flow values and data flow variables. The following rules deal with function application expressions. For the sake of simplicity, we assume that function application expressions have a single parameter only but all rules can be extended to the n-ary case in the straight-forward way.

Constant Folding (CF)

$$c_1 \sqcap_L c_2 \xrightarrow{CF} c_3 \qquad \text{with} \qquad c_3 = c_1 \sqcap_L c_2$$

The constant folding reduction replaces two constant terms by their safe approximation. It ensures that a single constant will remain on one level in the nesting structure of each expression.

Duplicate Variable Removal (VAR)

$$x \sqcap_L x \xrightarrow{VAR} x$$

The VAR-reduction reduces the occurrences of a single variable to a single representative. It is justified by the fact that the conservative approximation operator $\sqcap_L$ is reflexive.

Bottom Shortcut (BSC)

$$e \sqcap_L \bot \xrightarrow{BSC} \bot$$

The BSC-reduction exploits the special status of the least element $\bot$ in the inducing lattice. This element represents the loss of all information. No matter to which concrete lattice element the expression e evaluates, the final result of the conservative approximation with $\bot$ will always yield $\bot$. Therefore, the original compound expression can be represented much more efficiently by $\bot$ which is known to be the result of any possible evaluation.

The tuple representation is vital for the effectiveness of the BSC-reduction. It is much more likely that data flow information is lost for a single variable than for the whole program state.

Push Out Upper Bound (POUB)

$$\text{If} \quad [t(p)]|_{[x_i:=\top]} \sqcap c_{old} = c_{new} \sqsubset c_{old} \quad \text{then} \quad t(p) \sqcap c_{old} \xrightarrow{POUB} t(p) \sqcap c_{new}$$

The intuition of the POUB-reduction can be summarised as follows: even though we do not know the precise semantics of elementary transfer functions, we can still determine an *upper bound* for the expression $t(p)$. The reason is that the substitution of all variable occurrences in the parameter expression p with the most optimistic element leads to an upper bound for this expression and the result of the function application to such an upper bound is an upper bound of the function due to the monotony of t.

Intuitively, the reduction rule states that we can use the upper bound of an application expression to weaken the upper bound of the surrounding approximation expression. It's main purpose is to enable additional BSC-reductions. For example, consider an elementary transfer function which maps the most pessimistic element $\bot$ to itself - which is quite often the case. Then

$$\begin{aligned} t(e|_{[x_i:=\top]}) \sqcap c \quad &\xrightarrow{POUB} \quad t(e) \sqcap \bot \\ &\xrightarrow{BSC} \quad \bot \end{aligned}$$

Furthermore, the $\xrightarrow{POUB}$ also integrates the application of elementary transfer functions into the normalisation process but avoids a subtle challenge which arises, if a straight-forward application rule would have been integrated in the normalisation process.

Distributivity (DSTR)

$$t_i(p_1) \sqcap_L t_i(p_2) \quad \xrightarrow{DSTR} \quad t_i(p_1 \sqcap_L p_2)$$

The distributivity rule ensures that each normal form has a single application of a specific function on each level of the nested expression structure. Furthermore, it enables additional normalisations of the combined parameter expressions.

Obviously, we would like to integrate a rule like $t(c_1) \xrightarrow{APP} c_2$ which replaces an elementary transfer function which is applied to a constant by the result of the function application. Now assume that we perform a linear constant propagation which specifies the semantics of the increment operator ++ by an elementary transfer function `incr`. The problem arises at the join point in the example program depicted in Figure 5.10. The definition of the function meet

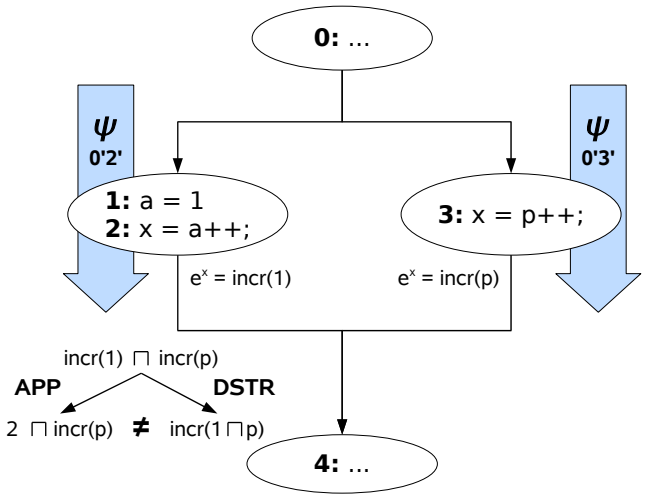


Figure 5.10: Interference of an APP- and the DSTR-rule

yields a safe expression for the evaluation function of x to incr(1) $\sqcap$ incr(p). This expression is subject to normalisation. Both the potential $\xrightarrow{APP}$-rule and the $\xrightarrow{DSTR}$-rule can be applied but the result expressions differ structurally! As a consequence, the expressions cannot be compared to each other easily, if the analysis phase and the validation phase apply the normalisation rules in a different way. A comparison operation would have to be aware of the fact that $2 \sqcap$ incr(p) and incr($1 \sqcap p$) are *semantically equivalent*. Such a check is clearly more complicated than the pure structural comparison we strive for.

This observation can even be stated more generally: any function representation has to supply a comparison operation that can compare functions independently from the way they are constructed by the analysis and the validation phase. This can be difficult if function operations like normalisation can yield different function representations which are semantically equivalent. The generic normalisation rules defined in this section solve this issue independently form the inducing data flow problem.

For example, the combination of the $\xrightarrow{POUB}$- and the *DSTR*-rule to safe approximation expression incr(1)$\sqcap$incr(p) yields $2 \sqcap incr(1 \sqcap p)$ independently of application order of the normalisation rules. This is a consequence of the uniqueness of the normal form which is proved in Section 5.3.2.

Interpretation

Data flow expressions define summary functions which map the program state from one point to another. The summary functions comprise the effects of all execution paths between the program points. Join points of paths lead to safe approximation expressions and function composition is realised by variable substitution. This way, a single data flow expression encodes the data flow which finally yields a specific value in the output state of a summary function.

The partial evaluation strategy compresses this information about the data flow through different paths in the program. Constant folding combines two data flow values which have been generated on different paths. Similarly, the reduction of the several occurrences of a variable x in a approximation expressions conflates the fact that the same piece of the input state - namely the value of variable x influences the output value in the same way on different paths.

The $\overset{BSC}{\longrightarrow}$-rules automatically drops dependencies which cannot influence the result. The element $\bot$ is the least informative element of the inducing data flow so that an expression $e \sqcap_L \bot$ states the fact, that the most pessimistic assumptions had to be made on one path. Thus, it does not matter what happens on any other path so that the expression e which describes the influence of the other path can be dropped without loss of precision.

The other rules deal with function application expressions which can be considered as an explicit formulation of the composition of two subsequent paths. The preceding path supplies the parameter expressions while the function application expressions comprises the effects of a single instruction, which could not be expressed by simpler elements of the model.

The $\overset{POUB}{\longrightarrow}$-rule deals with application expressions which cannot be evaluated because some of their parameter expressions still contain a variable. In such a case, we assume that the relevant piece of the input state which is denoted by variable x is the *best* of all possible data flow values - the most optimistic element $\top$. Then it is possible to evaluate the parameter expressions and to apply the elementary transfer function. This yields an *optimistic upper bound* for the application expression. No matter which input state is used, the application expression will always evaluate to an element that is not better than the upper bound. If this optimistic bound is *weaker* than the constant bound on another path, then we can reduce the constant expression without loss of precision. Essentially, the rule exploits the monotony of elementary transfer functions to inspect the potential influence of an function application expression. It is especially useful for function application expressions, which take more than one parameter. If one of the parameters is the most pessimistic element and another one still depends on the input state, then it is likely that the evaluation of the application expressions also yields the most pessimistic element.

5.3.2 Properties of Data Flow Expressions

It is vital for the validation process, that the validator can compare summary functions to each other. We achieve this in the following way: The summary function representation is reduced to data flow expressions which have a unique normal form. Furthermore, data flow expressions exhibit a specific syntactic structure which defines an order relation on data flow expressions. This order relation on expressions extends to an order relation on summary function. Thus, two summary functions can be compared with each other by reducing their defining data flow expressions to their normal form and subsequently compare the syntactic structure of the expressions.

We show in this Section that the reduction rules defined in Section 5.3.1 yield a unique normal form and define a partial order on the syntactic structure of data flow expressions in normal form.

Uniqueness of the Normal Form

The uniqueness of a normal form is an immediate consequence of the termination and the local confluence of the reduction relation. Intuitively, local confluence ensures that two intermediate expressions, which result from the application of two different reduction rules can be reduced to a common expression again. Additionally, locally confluent reduction relations which terminate reduce a term to a unique normal form independently from the order in which reduction rules are applied.

Lemma 1 (Termination of the Reduction Relation) *The reduction relation $\rightarrow_E$ terminates.*

Proof 1 *We can connect each expressions to a tuple $(n, c) \in \mathbb{N}_1 \times L$ which combines the number of subexpressions n and the most pessimistic constant c in the subexpressions of a specific nesting level.*

The point-wise extension of the order relations $\langle \mathbb{N}_1, \leq \rangle$ and $\langle L, \sqsubseteq_L \rangle$ yields a partial order on these tuples which is well founded if the partial order $\sqsubseteq_L$ is well founded. All reduction rules either decrease n or weaken c (POUB) so that there cannot be an infinite decreasing chain in $\langle E, \rightarrow_E \rangle$ because there is no infinite decreasing chain in $\langle \mathbb{N} \times L, \leq \times \sqsubseteq \rangle$

Lemma 2 (Local Confluence of the Reduction Relation) *The reduction relation $\rightarrow_E$ is locally confluent.*

Proof 2 *By finding locally confluent reduction sequences for each possible pair of elementary reductions. (See Appendix A)*

Theorem 2 (Uniqueness of the Normal Form) *The normal form $e\downarrow$ of an expression $e \in E$ with respect to the reduction relation $\rightarrow_E$ is unique.*

Proof 3 *The reduction relation $\rightarrow_E$ terminates due to Lemma 1, and it is locally confluent due to Lemma 2. Therefore, the reduction relation is confluent due to the Lemma of Newman[New42] and reduces to a unique normal form.*

The termination proof already shows, that the normalisation rules try to reduce data flow expressions to a minimal number of subexpressions. This is vital to keep the summary function representation compact and simplifies the comparison of summary functions which depends on the structural comparison of expressions in normal form.

Structure of Irreducible Expressions and a Checkable Order Relation

Now we define a partial order based on the structure of the normal form of expressions. Firstly, we prove that each expression in normal form either is the most pessimistic expression $\perp$ or it contains at most one constant, and each data flow variable as well as each application expression occurs only once within each of the nested subexpressions.

We assume that each each elementary transfer function t_i and each data flow variable x_k can be identified with a unique index i and k respectively. Let TI and VK denote the subsets of the corresponding index sets.

Once again we assume that each elementary transfer function takes a single variable as an argument to simplify the notation, but the arguments hold for the n-ary case as well.

Theorem 3 (Structure of Irreducible Expressions) *Let $e\downarrow$ denote the normal form of an arbitrary expression e. Then expression $e\downarrow$ has the following structure*

$$e\downarrow = \bigsqcap_{i \in TI} t_i(p_i) \sqcap_L \bigsqcap_{k \in VK} x_k \sqcap_L c$$
$$or \quad e\downarrow = \perp$$

and all parameter expressions p_i are also in normal form.

Proof 4 *By contradiction: Assume that e contains more than one occurrence of function application expression t_i, of variable x_k, or two constants. Then there is a reduction rule and e is not in normal form.*

Next, we define an order relation on expressions in normal form. Essentially, the order relation is a more elaborated variant of the simple order relation that suggests to consider expressions to be weaker which consists of strictly more subexpressions (see Section 5.1.2). The new order relation considers expressions to be weaker whose *normal form* contains strictly more subexpressions. Additionally, the order relation of the constant expressions is additionally considered for two expressions which contain the same set of other subexpressions.

Obviously, this order relation can be checked easily by the comparison of the structure of expressions in normal form.

Definition 5 (Order Relation of Expressions) *Let $e_1, e_2 \in E$. Then the* order relation of expressions *is defined as: $e_1 \sqsubseteq_{E\downarrow} e_2$ iff for $e_1 \downarrow$ and $e_2 \downarrow$ holds either $e_1 \downarrow = \bot$ or*

1. $TI_1 \supseteq TI_2 \wedge \forall i \in TI_2 : p1_i \sqsubseteq_{E\downarrow} p2_i$ *and*
2. $VK_1 \supseteq VK_2$ *and*
3. $c_1 \sqsubseteq_L c_2$

Theorem 4 *The relation $\sqsubseteq_{E\downarrow}$ is a partial order.*

Proof 5 *By induction over the structure of the expressions and the fact that the subset relation $\subseteq$ and the meet-operator of the inducing data flow lattice $\sqsubseteq_L$ are partial orders.*

The definition of the partial order yields a definition of the meet of expressions $\sqcap_{E\downarrow}$ as usual:

Definition 6 (Meet of Expressions) *Let $e_1, e_2 \in E\downarrow$. Then the* meet of expressions *is defined as $e_1 \sqcap_{E\downarrow} e_2 = e_3$ iff*

1. $e_3 \sqsubseteq_{E\downarrow} e_1 \wedge e_3 \sqsubseteq_{E\downarrow} e_2$
2. $\nexists e_4, e_4 \sqsupset e_3 : e_4 \sqsubseteq_{E\downarrow} e_1 \wedge e_4 \sqsubseteq_{E\downarrow} e_2$

Theorem 5 (Property of the Meet of Expressions) *The meet of expressions can be modelled by the meet operation of the inducing lattice. It holds that*

$$e_1 \sqcap_{E\downarrow} e_2 = e_3 = e_1 \sqcap_L e_2$$

Proof 6 *By comparison of the normal forms of e_1, e_2, and e_3 to establish $e_3 \sqsubseteq_{E\downarrow} e_1$ and $e_3 \sqsubseteq_{E\downarrow} e_2$ and proving the maximality of e_3 by contradiction.*

See Appendix A.

Thus, the safe approximation $\sqcap_{E\downarrow}$ which combines two expressions with the safe approximation operator of the inducing lattice, forms itself a lattice on the set of data flow expressions.

Evaluation of Expressions

Next, we show that expressions, which do not contain any function variables, preserve the lattice order of the inducing lattice L whenever they are evaluated with variables replaced by elements of the inducing lattice. We do not formally define the semantics of the rather intuitive evaluation process at this point, but remark that it seems to be closely related to *Herbrand interpretation* of arithmetic expressions as used in [RKS99], [MORS05].

Definition 7 (Applicable Expressions) *Let $E_{app} \subset E$ denote the set of data flow expressions which do not contain free function variables. We call an expression $e \in E_{app}$ an* applicable *expression.*

Free function variables are only required to model modular results (refer to Section 5.4). They occur only as intermediate results during the computation of the final interprocedural summary functions. Only the final summary functions need to be evaluated, so that we limit the definition of the evaluation operation to *applicable* expressions.

Lemma 3 (Evaluation of Applicable Data Flow Expressions) *Then*

$$\forall v \in L, e_1, e_2 \in E_{app} : e_1 \sqsubseteq_{E\downarrow} e_2 \Rightarrow e_1|_{[x:=v]} \sqsubseteq e_2|_{[x:=v]}$$

Intuitively, weaker expressions with respect to the order relation of expression always evaluate to weaker data flow values.

Proof 7 *By induction over the structure of (applicable) expressions and the fact that weaker expression can only contain:*

1. *additional terms in conservative approximation expressions*
2. *monotone function applications which operate on weaker or equal parameters*
3. *weaker or equal constants.*

This lemma is the central prerequisite for the definition of summary function operations.

Remark

The discussion in Section 5.2.2 already remarks that function application expressions induce nested expressions and that the nesting depth is not bounded if a function application expression is subsequently used as its own parameter expression due to cycles in the flow graph. This phenomenon can now be interpreted in the algebraic model.

The order relation of expression is a partial order. However, it is not a *well founded* partial order. The normalisation rules guarantee that any normal form of an expression has a limited number of subexpressions on each nesting level but the nesting depth is not limited. In contrast, it can be infinite due to the recursive definition of function application expressions which allow arbitrary parameter expressions.

As a consequence, we cannot show by a simple argument that any data flow analysis which involves the expression lattice terminates. A safe - but overly conservative - way to deal with this issue is to limit the maximum nesting depth to a constant number n and to approximate all parameter expressions at this level by the safe lower bound $\perp$. This fits smoothly into the expression model but potentially decreases the precision of an analysis which uses the model.

5.3.3 Properties of the Summary Function Model

The functional approach to interprocedural analysis uses summary functions in two different ways. Firstly, they are the data flow values during the summary function computation phase. Secondly, they act as transfer functions during the value computation phase where they map the safe approximation of an invocation context of a method to the context of each call-instruction in the method.

Thus, summary functions have to exhibit two different kinds of properties to be usable in the different phases. The first phase requires that is is possible to specify the computation of summary functions in terms of a data flow problem. Essentially, this requirement can be broken down to two elementary properties:

- The set of summary functions has to form a lattice with respect to an order relation and the function meet defined for the summary functions.
- Function composition with a specific instruction-level summary function has to be monotone. This is necessary because the function composition operation acts as transfer function in the function computation phase.

The fact that the summary function computation can be expressed as a data flow problem ensures, that we can validate given summary functions by the general validation principle.

The summary functions, which are computed in the functional phase, act as transfer functions during the computation of safe approximations of invocation contexts. Therefore, they have to exhibit the following additional property

- The application of summary functions has to be monotone with respect to elements of the inducing lattice L.

We will now prove that these properties hold for *applicable summary functions*. Applicable summary functions are defined as follows:

Definition 8 (Applicable Evaluation Functions) *We call an evaluation function* applicable *if its defining expression does not contain any function variable application expression.*

Definition 9 (Applicable Summary Functions) *We call a summary function* $\psi \in \Psi$ applicable *if all of its evaluation functions are applicable. We denote the subset of applicable summary functions by* $\Psi_{app} \subset \Psi$.

Essentially, an applicable summary function does not contain free function variables. The reason for the separation is that function application substitutes data flow variables and evaluates the underlying expressions which is not possible as long as a data flow expression does not contains function variables.

We call summary functions *open* if they contain function variables because function variables express unresolved dependencies to external code. This representation is necessary to encode the analysis results for separated software modules in a flexible way. The extension is discussed in Section 5.4.2. The

general difference between applicable and open summary functions is that only applicable summary functions can act as transfer functions in the value computation phase. However, we show in Section 5.4.2 that the computation of open summary functions can also be expressed as a data flow problem so that a validator can check open summary functions according to the general validation principle as well. The open summary function representation increases the flexibility of the analysis and validation phase so that it is possible to deal with the separate analysis of software modules in various ways.

We now show that applicable summary functions exhibit the required properties. The definition of summary functions directly depends on data flow expressions. Particularly, function meet, function comparison, and function application reduce to the meet, comparison and evaluation of data flow expressions. Thus, the general idea is to reduce the properties of summary functions to the properties of expressions. The central prerequisite is Lemma 3 which ensures that the evaluation of data flow expressions preserves the order of values of the inducing lattice.

Partial Order on Summary Functions

Firstly, we refine the intuitive definition of summary function application given in the introduction of the function model. The important additional aspect is, that we have to restrict the definition of function application to applicable summary functions because the defining expressions cannot be evaluated properly if they still contain function variables. Once again, we restrict ourselves the single variable case, i.e. $Var = \{x\}, env_m = \langle x \rightarrow x_m \rangle$ to simplify the notation. The extension to full environments is straight forward.

The application of summary function is reduced to variable substitution in the defining expressions by

Definition 10 (Application of Summary Functions) *Let $\psi \in \Psi_{app}$ an applicable summary function and $e \in E_{app}$ its defining expression. Then*

$$\forall v \in L : \psi(v) =_{df} (e^x|_{[x:=v]})$$

Next, we define an order relation on summary functions by a reduction to the order relation we have specified for data flow expressions. In this section we restrict ourselves to applicable summary functions.

Definition 11 (Order of Summary Function) *Let $\psi_1, \psi_2 \in \Psi_{app}$ and let $e_1, e_2 \in E_{app}$ be their defining expressions. Then*

$$\psi_1 \sqsubseteq_\Psi \psi_2 =_{df} e_1 \sqsubseteq_{E\downarrow} e_2$$

Our final goal is to show, that summary functions form a lattice with respect to the specified order relation. Therefore, the order relation has to be a partial order. A summary function is considered to be more conservative than another, if it maps all elements of the domain to a result element which is at least as conservative as the corresponding result of the second function. Thus,

Theorem 6 (Partial Order of Applicable Summary Functions) *The order is a partial order on applicable summary functions with respect to evaluation in L i.e.*

$$\psi_1 \sqsubseteq_\Psi \psi_2 \Rightarrow \forall v \in L : \psi_1(v) \sqsubseteq_L \psi_2(v)$$

Proof 8 *Immediate consequence of Definition 11 and Lemma 3.*

Essentially, the fact that the evaluation of data flow expressions preserves the order relation of the inducing lattice directly implies that the application of summary functions preserves the order relation as well. Similarly, the meet of summary functions is in fact a meet operation because it is also reduced to the meet of expression. Thus, summary functions form a lattice with respect to the order relation induced by the order relation of expressions.

Monotony of Function Composition

Next, we have to show, that function composition with a fixed summary function is monotone. This is necessary, because function composition with instruction-level summary functions defines the transfer functions of the functional data flow problem. Function composition is reduced to the substitution of data flow variables in the first function by the defining expressions of the second function. Thus,

Definition 12 (Function Composition) *Let $\psi_1, \psi_2 \in \Psi_{app}$ and $e_1, e_2 \in E_{app}$. Then we define the* function composition *as follows:*

$$\psi_1 \circ \psi_2 =_{def} e_1|_{[x:=e_2]}$$

This definition of function composition is monotone with respect to the order relation of summary functions. We consider the function composition with a fixed summary function ψ_c which models the semantics of a single node in the flow graph.

Theorem 7 (Monotony of Function Composition) *The composition of applicable summary functions is monotone in $(\Psi, \sqcap_{psi})$. Let $\psi_c \in \Psi_{app}$:*

$$\forall \psi_1, \psi_2 \in \Psi_{app} : \psi_1 \sqsubseteq_\Psi \psi_2 \Rightarrow \psi_c \circ \psi_1 \sqsubseteq_\Psi \psi_c \circ \psi_2$$

Proof 9 *For the order relation on summary functions to hold, the order relation on their defining expressions has to hold. Thus, with the definition of the composition we can reduce the proposition to:*

$$\forall e_c, e_1, e_2 \in E_{app} : e_1 \sqsubseteq_{E\downarrow} e_2 \Rightarrow e_c|_{[x:=e_1]} \sqsubseteq_{E\downarrow} e_c|_{[x:=e_2]}$$

which is clearly the case according to the definition of $\sqsubseteq_{E\downarrow}$ (see Definition 5).

The important observation is that the order relation of expressions ensures that weaker expressions contain at least the same subexpressions as stronger ones. Thus, they contain at least the same data flow variables. As a consequence, the substitution of variables with specific expressions always yields expressions which contain at least the same expressions again.

All in all, we have shown, that the computation of summary functions can be expressed in terms of a data flow problem. Thus, we can apply the general validation principle to check their validity at the consumer side.

This property holds independently from the inducing data flow problem because the summary function model relies on generic properties of the inducing lattice only.

Monotony of Summary Functions

The summary functions, which have been computed in the functional phase, act as transfer functions in the subsequent analysis phase which computes a safe approximation for the invocation context of each method.

It is a prerequisite that transfer functions of a data flow analysis are monotone with respect to the order relation of the lattice of data flow values.

Theorem 8 (Monotony of Applicable Summary Functions) *The applicable summary functions are monotone in* $(L, \sqcap_L)$.

$$\forall v_1, v_2 \in L : v_1 \sqsubseteq_L v_2 \Rightarrow \psi(v_1) \sqsubseteq_L \psi(v_2)$$

Proof 10 *Immediate consequence of Definition 10 and Lemma 3.*

Thus, it is in fact possible to use summary functions which are expressed in terms of the expression model as transfer functions for the second phase. Together with the preceding result we come to the conclusion that it is possible to validate summary functions and to use them in the subsequent value computation phase. Thus, the summary function model allows to deal with the validation of the functional part of the interprocedural analysis in a way that does not depend on the inducing analysis.

5.3.4 Summary Functions and the Inducing Data Flow Problem

The summary function model inherits the properties of the inducing data flow problem. Summary functions are monotone with respect to order of the inducing data flow lattice because the evaluation of data flow expressions involves the application of monotone transfer functions of the inducing problem and the conservative approximation operator of the inducing lattice only.

Similarly, the comparison criterion which investigates the structure of data flow expressions on a syntactical level depends on the fact that only the conservative

approximation operator can combine different subexpressions which each other. Therefore, additional subexpressions can only weaken the result of the valuation of an expression which is vital for the prove that the order relation of summary functions has the necessary properties.

The normalisation of expressions splits data flow expressions into equivalence classes. The uniqueness of the normal form guarantees that there is a distinguished element for each equivalent class which can act as the representative for the comparison operation. Furthermore, the normal form is more compact than arbitrary expressions because the reduction rules mimic the behaviour of a partial evaluation of the expression.

However, the distributivity reduction, which reduces all function application expressions to a single application expression on each level in the nesting structure, demands that the inducing data flow framework is distributive. Therefore, non-distributive problems like integer constant propagation which uses symbolic evaluation of arithmetic expressions cannot be handled properly by the normalisation process. The reason is that the "early meet" has the potential to loose precision for non-distributive functions. It is possible, that

$$t(a \sqcap b) = c_1 \sqsubset c_2 = t(a) \sqcap t(b)$$

As a consequence, the uniqueness of the normal form can no longer be guaranteed because it depends on the order of $\overset{DSTR}{\longrightarrow}$ and $\overset{POUB}{\longrightarrow}$-reductions.

There are two different ways to approach the challenge which arises from the non-distributivity of an inducing analysis: Firstly, the evaluation order of the analysis and validation phase can be "synchronised" so that the loss of precision always occurs at the same points. However, this removes an important degree of freedom and it requires additional formal justifications. The second approach is that the analysis phase computes the results which states the *maximal* loss of precision due to non-distributivity. The idea is to meet all analysis results as early as possible in order to compute a fix-point which conservatively approximates the results of all possible evaluation sequences. The intentional under-approximation principle guarantees that the validator can validate this fix-point independently from its own evaluation sequence.

We do not consider this aspect further, because the demand to support the validation of non-distributive analyses has not arisen yet. Furthermore, the potential impact on the efficiency of the function representation or the quality of the results might be significant.

5.4 Modular Results and Incremental Validation

One of the requirements which are identified in Chapter 4 is that the validation process shall support an *incremental validation scenario*. This scenario expects that the whole program is structured into several modules each of which is shipped to the consumer site in isolation.

One of the important prerequisites is the ability to express the data flow results for a single software module in a flexible way. Essentially, we want to be able to

1. derive a *safe lower bound* for the analysis results of each available module at any point in time and to
2. *compose* the representation of results from different modules in order to construct a result for a larger part of the program.

This capability is useful for both the analysis and the validation phase. The analysis phase can use the flexible representation of a modular result to apply a number of strategies that deal with the influence of external code in different ways. We discuss this in more depth during the presentation of our evaluation methodology in Chapter 9.

The validator can use the representation for modular results in two different ways. Firstly, the validator can compute a valid lower bound for analysis results at any point in time. Thus, it is possible to apply optimisation which depend only on the safe lower bounds immediately. Secondly, all pieces of the whole program analysis result which coincide with the safe lower bound can be considered valid. Thus, the safe lower bound acts as a checking criterion for the validity of data flow facts and separates closed from open data flow facts. The validator can subsequently check the validity of each modular result and it can combine the modular results into a whole program result. The primary goal is to increase the efficiency of the validation process in an *incremental validation scenario*.

The first question is, which granularity to choose for modular data flow results. The granularity of the results determines the minimal scope of a modular result. If this scope is small, then the validator is able to partially use the data flow results early and intermediate results which are only relevant for the validation of the module can be dropped. Similarly, the analysis can estimate the potentially effects of external code in a more fine-grained manner on a small scope.

We choose a single method as the minimal part of the program that is considered in isolation for two reasons. Firstly, a method is the key abstraction of the functional approach to interprocedural analysis. Therefore, the representation of modular results fits smoothly into the analysis model. Secondly, a single method is a natural scope for the early use of analysis results for example in an optimisation scenario. An incremental validation or analysis on a per method basis can trigger at least some optimisations immediately after the code of the method has been considered.

The next question is how external code can influence the analysis result which is derived from the context of a single method. The analysis results of a method depend on the analysis results from the rest of the program in two different ways. Firstly, other methods can call the method under consideration. Each call provides a new invocation context for the method which can weaken the assumptions about the program state at the start of the method. Secondly, the

method can itself call other methods. Such a call influences the assumptions about the program state immediately after the corresponding call instruction.

This section explains how the model of summary functions has to be extended in order to represent modular results. Section 5.4.1 considers the influence of external calls on the invocation context of the method under consideration. It turns out that the summary function model is already inherently able to deal with this issue. The following section investigates the influence of the callees of the method under consideration. The central modelling idea is to capture this dependency by the introduction of free function variables into the expression model. This way the composition of modular results boils down to the substitution of function variables. Furthermore, the introduction of function variables yields a mechanism to estimate the potential effects of external code by the substitution of different kinds of summary functions. Section 5.4.3 illustrates the application of this technique by an interesting observation: any intraprocedural analysis result is in fact a safe approximation of the modular result for the method under consideration.

Section 5.4.5 discusses the formal properties of function variables in the expression model. Finally, we conclude with a reinterpretation of the incremental validation process for the extended function model.

5.4.1 Invocation Contexts and Data Flow Variables

The influence of the invocation context of a method shows up in the definition of the intermediate program states I_i and O_i before and after the execution of the instruction i in a method m. According to the definition of interprocedural data flow problems in Section 4.2 it holds that

$$\forall i \in \mathit{FlowNodes}_m : I_i = \psi_i(IC_m) \wedge O_i = \psi_{i'}(IC_m)$$

Each intermediate state I_i and O_i can be computed from the intraprocedural summary functions $\psi_i, \psi_{i'}$ which map the state at program point 0 given by IC_m directly to the intermediate state.

The following equation defines the invocation context IC_m and reveals the dependency on call sites in other methods.

$$IC_m \sqsubseteq \bigsqcap_{n \in \mathit{CallSites}(m)} I_n$$

Essentially, the assumptions about the invocation context IC_m of a method m have to subsume the assumptions about the program state I_n at each call site. Consequently, no valid information about an invocation context can be derived until all possible call sites are available. This renders all invocation contexts of publicly accessible methods open until all software modules have been transferred to the consumer.

Nevertheless, it is possible to exploit specific language features to detect that new code cannot contain additional calls to a specific method. For example, private methods in Java cannot be called directly from code outside the defining class. Thus, all call sites of private methods are known after a class is loaded completely. However, even the invocation context of private methods indirectly depends on the invocation context of some public method which acts as an entry point for the control flow into the software module. Thus, the effectiveness of the computation of invocation contexts is limited within a single software module.

Thus, it is difficult to establish the validity of invocation contexts if additional pieces of code can still be transmitted to the consumer. Therefore, it is important that the validator can safely approximate the potential effects of invocation contexts at any time during the validation process. This can be achieved easily, if the validator uses safe assumptions about an invocation context. The most pessimistic element of the inducing lattice is always a safe lower bound because it states that nothing is assumed about the program state at all. However, some analysis provide better, problem specific lower bounds, which can be used in the same way than the most pessimistic one.

The use of the most pessimistic element as a lower bound for an invocation context IC_m yields a safe lower bound for each intermediate states within method m.

$$I_i^{\perp} = \psi_i(\perp) \sqsubseteq I_i \quad \wedge \quad O_i^{\perp} = \psi_{i'}(\perp) \sqsubseteq O_i$$

The result states $I_i^{\perp}$ and $0_i^{\perp}$ safely approximate the states from the analysis result because the intraprocedural summary functions ψ_j are monotone in L. Even if the validator uses the most pessimistic element $\perp$ as a safe lower bound for the invocation contexts then the safe lower bounds for the intermediate states can provide a significant amount of information.

Consider the example code in Figure 5.11 and assume that the analysis in question performs copy constant propagation. The invocation context IC_m consist of the values of the three parameters p_1, p_2 and p_3 of method m. Obviously, the assumptions about the invocation context - i.e. the question whether a parameter always holds a constant value - depends on the method calls of m throughout the program.

The assumptions about the program state immediately after the execution of instruction 4 are captured by the output state O_4. This state is computed by the intraprocedural summary function $\psi_{4'}$ which directly maps the state before the first instruction in node 0 - namely the invocation context - to the state O_4. The summary function $\psi_{4'}$ is the composition of the instruction-level summary functions of the instructions 0,1,2, and 4. This composition yields the following summary function

$$\psi_{4'} = \langle e_{4'}^{p1}, e_{4'}^{p2}, e_{4'}^{p2} \rangle = \langle 5, 2, p_3 \rangle$$

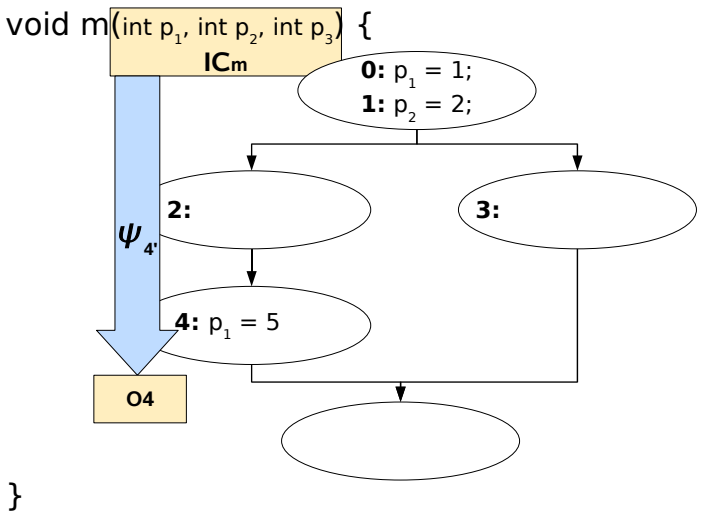


Figure 5.11: Safe Approximation of Invocation Contexts

Essentially, this summary function states that the value of p_1 will always be constant 5 after instruction 4, and that the value of p_2 will always be constant 2. The value of p_3 depends on the value of the parameter p_3 in the invocation context.

As stated before, the validator can derive a safe lower bound for the state O_4 from a safe lower bound for the invocation context IC_m. If the validator chooses the most pessimistic element $(\bot, \bot, \bot)$ as lower bound, then the intraprocedural summary function $\psi_{4'}$ yields the safe lower bound $O_4^{\bot}$ as

$$O_4^{\bot} = \psi_{4'}(\bot, \bot, \bot) = (5, 2, \bot)$$

This value is a safe approximation of the data flow result and it is valid *independently from* the value of the invocation context IC_m. Furthermore, the assumptions about the program state 0_4 is significantly more informative than the most pessimistic assumption would have been.

This way, the validator can partially use the analysis result immediately after the inspection of method m even though the validity of the whole solution cannot be established yet.

The same technique can be applied by the analysis phase, too. If the closed world assumption does not hold, then the analysis can consider the publicly visible methods as additional entry points because they can be called by external code. Thus, the analysis has to expect some unknown call site, which supplies a pessimistic invocation context, so that it has to conservatively approximate the corresponding invocation contexts. However, the analysis can still try to determine more precise invocation contexts for methods, which are not accessible from outside the software module.

The example reveals, how the dependencies between intermediate states and the invocation context are encoded in the summary function model. The intraprocedural summary functions refer to the parameter environment - which is the invocation context of the method - by data flow variables. For example the value of p_3 at point 4' is the same as the value of p_3 in the invocation context. In contrast, constant expressions evaluate to more precise values, even if the most pessimistic element $\perp$ is substituted for the variable values. This way the basic definition of the summary function model already supports the computation of safe lower bounds for intermediate states and does not require any extensions.

However, the strategy in general requires, that the intraprocedural summary function like $\psi_{04'}$ are valid. Unfortunately, they can depend on summary functions which capture the semantics of unknown callees. Such summary functions cannot be validated without the code of the callee. Section 5.4.2 defines an extension of the summary function model, which allows to derive a safe lower bound for summary functions. Such a summary function may be less precise than the final one but we can use it to derive valid lower bounds for the intermediate states even without knowledge of the whole program.

5.4.2 External Callees and Function Variables

The method under consideration can call other methods. The summary functions of these methods contribute to the intraprocedural summary functions of the caller and it is not possible determine the callee summaries, without code of the callees.

Like in the previous section, we take a look at the relevant equations in the definition of a data flow solution (refer to Section 4.2.5). Intraprocedural summary functions are defined in the following way:

$$
\begin{aligned}
\psi_{i'} &\sqsubseteq f_i(\psi_i) \qquad \text{with} \begin{cases} \text{for} i \notin \mathit{Call}: & f_i(x) = \psi_{ii'} \circ x \\ \text{for} i \in \mathit{Call}: & f_i(x) = \psi_{\mathit{call}_n} \circ x \end{cases} \\
\psi_i &\sqsubseteq \bigsqcap_{j \in \mathit{pred}(i)} \psi_{j'} \\
\psi_m &\sqsubseteq \psi_{\mathit{Exit}_m}
\end{aligned}
$$

Essentially, the intraprocedural summary function that maps the invocation context to the state at program point i' after instruction i is defined by the composition of the intraprocedural summary function of the point immediately before the execution of the instruction and the instruction-level summary function $\psi_{ii'}$. The situation differs for call instructions. At invocation sites, the composition involves the *interprocedural* summary function ψ_{call_n} of the callee n. This can only be determined if method n is available.

As a consequence, all intraprocedural summary functions of the current method which depend on unavailable callee summaries cannot be determined completely, too. In order to deal with this issue we introduce a second representation of intraprocedural summary functions which contains all dependencies on unknown summary functions in terms of *free function variables*. This representation is flexible because we can substitute function variables either to derive safe lower bound or to integrate summary functions of the callees as soon as they become available during the analysis or validation phase.

During the validation process the code consumer can derive a safe lower bound for all program states in an available method by a combination of the safe approximation of the invocation context and the safe approximation of intraprocedural summary functions: the validator just replaces all free function variables by safe lower bounds for the corresponding callee summaries. The result is a safe lower bound for each intraprocedural summary in the caller which in turn can be used to derive a safe lower bound for each intermediate state.

In order to represent calls to external callees, we introduce function variable expressions into the data flow expression model:

Definition 13 (Variable Function Application Expression) *Let S be a set of function variables, $s_i \in S$ a function variable, and $e_1, \ldots, e_n \in E$ data flow expressions. Then the* variable function application expression $s_i(e_1, \ldots, e_n)$ *is a data flow expression.*

A function variable acts as a placeholder for a single evaluation function in the summary function of the callee. This evaluation function yields a single data flow fact in the output state of the callee which is represented by the function variable expression. The parameters in the function variable expression model the input state of the callee. The parameter expressions are required to integrate the callee summary if it becomes available. In order to simplify the discussion we will just say that a function variable refers to a specific callee summary without explicitly stating the specific evaluation function within the summary function.

This kind of function representation cannot be used directly as a transfer function for the computation of data flow values, because its definition depends on summaries of external callees. However, the function representation can be computed and validated without knowledge about the external callees. Furthermore, it can act as a skeleton for the a safe lower bound and for the solution candidate of the final summary. We can either substitute the function variables by safe lower bounds which yields a safe lower bound for the summary in question or it can integrate summary functions of the callees in order to derive a solution for the greater context which involves the new methods, too. Thus, the new function representation can be considered as an open summary function, which can be closed by substitution of the external summaries in various ways.

We define the following terminology in order to separate the new kind of summary functions from applicable summary functions as defined in Section 5.3.3.

Definition 14 (Open Data Flow Expressions) *We call a data flow expression $e \in E$* open *if it contains a function variable application expression. We denote the subset of open data flow expressions by $E_{open} \subset E$.*

Definition 15 (Open Evaluation Functions) *We call an evaluation function* open *if its defining expression is open.*

Definition 16 (Open Summary Functions) *We call a summary function $\psi \in \Psi$* open *if one of its evaluation functions is open. We denote the subset of completable summary functions by $\Psi_{open} \subset \Psi$.*

We discuss the properties of free function variables and the validation of open summary functions in Section 5.4.5. At this point, we just consider an illustrative example in order to provide a first intuition about the use of open summary functions.

The example in 5.12 is an extended version of the example which we use to consider the safe approximation of invocation contexts in Section 5.4.1. It additionally contains two calls at point 2 and 3 respectively.

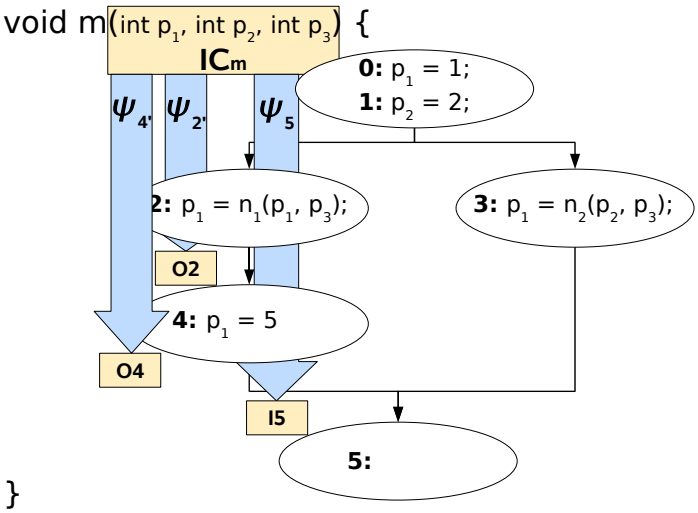


Figure 5.12: Open Summary Functions

The summary function $\psi_{2'}$ maps the invocation context to the program state in O_2. It is an open summary function because it still contains a function variable s_{n1}, which represents the potential effect of the unknown call to n_1.

$$\psi_{2'} = \langle e^{p1}_{2'}, e^{p2}_{2'}, e^{p3}_{2'} \rangle = \langle s_{n1}(1, p_3), 2, p_3 \rangle$$

The defining expression $e^{p1} = s_{n1}(1, p_3)$ is constructed during the function composition of $\psi_2 = (1, 2, p_3)$ and instruction-level summary function $\psi_{22'} =$

$\langle s_{n1}(p_1, p_3), p_2, p_3\rangle$. The instruction-level summary $\psi_{22'}$ captures the effects of the call instruction at point 3. The function variable s_{n1} acts as a placeholder for the evaluation function of the return value in the callee summary ψ_{n1}.

The invocation context of n_1 at point 2 corresponds to the values of p_1 and p_3 immediately before the execution of the call instruction. This is represented in the instruction-level function by the fact, that the function variable expression takes the data flow variables p_1 and p_3 as parameters.

These two variables are substituted by the constant expression 1 and the data flow variable p_3 during the function composition of ψ_2 and $\psi_{22'}$. Finally, the open summary function models, that the value of p_1 at point 2′ corresponds to the application of the summary function of n_1 to the program state $(1, p_3)$, where the second parameter of the call is supplied by the value of the third parameter p_3 from the invocation context of m.

The summary function $\psi_{4'}$ which maps the invocation context IC_m directly to the state O_4 shows an interesting phenomenon. The function composition of $\psi_{22'}$ and the instruction-level summary of instruction 4 yields

$$\psi_{4'} = \langle 5, 2, p_3\rangle$$

which in turn states that the program state O_4 does *not* depend on the call of method n_1. This is reasonable, because we assume that the function call affects the value of local variable p_1 only, for which new data flow information is generated by instruction 4.

The subsequent join operation combines this summary function and the summary function ψ_5 which corresponds to the whole method summary ψ_m if the final node 5 does not change the data flow values.

$$\psi_m = \langle 5 \sqcap s_{n2}(1, p_3), 2, p3\rangle$$

This representation reveals valuable information about the dependency between method m and its callees. Firstly, the value of $p1$ at the end of the method invocation depends on the summary function of method n_2. In contrast, the values of p_2 and p_3 do not depend on any callee. Furthermore, the method invocation in point 3 influences intermediate program states in method m only, but it does not influence the summary function ψ_m. Thus, the validity of ψ_m does not depend on the validity of the summary function of method n_1. Therefore, the summary function ψ_m can be determined or validated even without any knowledge of ψ_{n1}.

Additionally, open summary functions provide a safe lower bound for their final counterparts. The summary function which always evaluates to the most pessimistic element of the inducing data flow analysis is a safe lower bound for any function variable expression. If we substitute all function variables with this summary function, then the result is a safe lower bound for the open summary function. This strategy yields

$$\psi_m^{\perp} = \langle \perp, 2, p_3\rangle$$

as a safe lower bound for interprocedural summary of the method m. The *final* summary function is equal to this lower bound if the analysis phase had concluded that the summary function of n_2 does never yield a constant value. Interestingly, the validation phase can establish the validity of the summary function ψ_m in such a situation without ever considering method n_2.

Furthermore, the safe lower bound of an open summary function is applicable because all function variables have been removed. Therefore, they can be used to derive safe lower bounds for intermediate states even if the states depend on external method invocations. For example, the evaluation of $\psi_2^{\perp}(\perp, \perp, \perp) = (\perp, 2, \perp)$ shows that the parameter p_2 always still has the constant value two after the execution of the method call to n_1.

5.4.3 Intraprocedural Analysis is an Application of the Safe Lower Bound Principle

The observations in Section 5.4.2 show how it is possible to model the results of a modular analysis in such a way, that the effects of external code can either be safely approximated or later substituted with more precise values.

Interestingly, intraprocedural analysis is a special case of the safe approximation strategy. An intraprocedural analysis aims at the computation of data flow facts which hold independently from the rest of the program. Similarly, the determination of a safe lower bound also safely approximates the potential effects of unknown pieces of code.

If you consider a single method in isolation, then the effects of internal parts of the method are twofold. Firstly, some unknown call site can provide an arbitrary invocation context for the method. Secondly, each call site within the method under consideration can affect the intermediate program state in the caller.

The safe lower bound principle safely approximates these two effects. Assume that an open representation for each intraprocedural summary function is available. These open representations contain the potential effects of external method calls on the result state in terms of function variable expressions. The safe lower bound computation substitutes these expressions by the most pessimistic summary function. The result is an applicable summary function, that is a safe approximation of the final summary function in the interprocedural counterpart of the analysis. It is safe because the use of the most pessimistic summary function ensures that nothing is assumed about any method invocation and it is usually an approximation, because an interprocedural analysis can provide a more precise summary of the callee.

The second effect of external code are the potential invocation contexts at all call sites. The second phase of the interprocedural analysis computes a safe approximation for all of these invocation contexts. This approximation contains assumptions about the program state that hold for each call site in the program. If a single method is considered in isolation, then nothing can be assumed about

the invocation context because some unknown call can happen in a program state where an assumption does not hold. Therefore, the determination of a safe lower bound for an intermediate state uses the most pessimistic element of the data flow problem to capture this situation. Thus, the evaluation of the safe lower bound of an intraprocedural summary function with the safe lower bound for the invocation context yields a safe lower bound for the intermediate state. This lower bound, does not depend on the behaviour of external callees and not on a precise invocation context either.

The intraprocedural variant of the specific analysis deals with the influence of unknown program parts *exactly* the same way. The definition of the intraprocedural problem requires the definition of a transfer function for each call instruction in the code. These transfer functions necessarily have to make worst-case assumptions about the modifications a callee can make. This is exactly the same as to model the influence in terms of function variable expressions and to safely approximate these expressions afterwards. Similarly, the intraprocedural analysis initialises the input state of the start node of the flow graph with assumptions about the invocation context which hold independently from the invocation sides of the method. This is the same, as to use a safe lower bound for the invocation context during the the approximation of intermediate results. All in all, the safe approximation of an open summary function result yields the results of the corresponding intraprocedural analysis.

The two approaches differ only with respect to the specification of instruction-level functions. An intraprocedural analysis does not compute intraprocedural summary functions but applies the transfer functions of the instructions of the methods during the propagation and computation of data flow values. Thus, only the application of transfer functions has to be specified. In contrast, the functional approach to interprocedural analysis subsequently connects instruction-level summary functions to larger functions. This requires function composition and function meet, which does not have to specified for the transfer functions of the intraprocedural analysis.

5.4.4 Open Summary Functions and the Incremental Validation Scenario

If we want to use the open summary function model during the validation process, then we have to be capable to validate open summary functions supplied by the analysis phase. This is possible because the computation of open summary functions for a software module is a data flow problem. Thus, we can apply the general validation principle even to the validation of modular results, as discussed in Section 5.4.5.

This observation gives rise to an extended validation scenario which deals with modular results. Assume that the validator receives two function representations for each method: an open one which describes the dependency on callee summaries and an applicable one where all of these dependencies have been resolved by the analysis phase. The validation of the open representation can

be performed immediately because it involves the inspection of the code within a method only. The validation of the applicable representation can be achieved as follows. If the open summary function does not contain a reference to other callee summaries any more, then it is valid and it can be safely substituted for all corresponding function variables in the open representations of the callers. This substitution strategy eventually validates all summary functions.

This strategy succeeds only if the call graph of the program forms a DAG. Any cycle in the call graph of the program introduces a self-dependence in the open representation of the summary function. At this point, the applicable function representations are required. They constitute a fix-point solution of the summary function computation. Especially, they do not contain self-dependencies anymore and serve as a "guess" for the correct solution of the recursive structure. Therefore, the validator just has to check that the substitution of the applicable summary function for the variables in the open representation is safely approximated by the corresponding applicable function.

All in all, the validator can incrementally compose the results from several software modules which is one of the key properties for the incremental validation scenario.

The second key property is that the validator shall be able to determine a safe lower bound for the available pieces of the result at any point in time. This is also immediately possible. All remaining variable function application expressions just have to be substituted by safe lower bounds of their result value. This effectively removes all function variables and turns an open representation into an applicable one which safely approximates the potential effects of the external call.

5.4.5 Properties of Open Summary Functions

Section 5.3.3 contains the formal justifications which ensure that applicable summary functions form a lattice with respect to the meet operation. Furthermore, function composition is shown to be monotone with respect the to the partial order of the function lattice. These properties are vital to argue, that the computation of applicable summary functions can be considered to be a data flow problem.

When we reconsider the formal line of argumentation for open summary functions we encounter a subtle problem: The proofs indirectly rely on the fact that data flow expressions preserve the order of the value lattice under evaluation (see Lemma 3). However, evaluation is *not* defined for open data flow expressions because they contain function variables.

Therefore, we prove an additional result in order to establish the bridge between applicable and open summary functions. The idea is to show that the substitution of function variables by applicable evaluation functions yields an applicable expression. Furthermore, the result expression preserves the order of the substituted evaluation functions in the sense that it evaluates to more

conservative results whenever a more conservative function is used for the substitution.

We start with a definition of function variable substitution. Once again we simplify the notation to the single variable x and remark that the extension to larger environment is straight-forward.

Definition 17 (Substitution of Function Variables) *Let $e = s(ep^x)$ be a function variable application expression with parameter expression ep and $L^{n=1} \rightarrow L : f(x) = ef^x$ an evaluation function with defining expression ef^x. Then the substitution of the function variable s by f denoted by $e|_{[s:=f]}$ is defined as:*

$$e|_{[s:=f]} = ef|_{[x:=ep^x]}$$

Interestingly, the definition corresponds to the definition of function composition (see Definition 12) which also substitutes data flow variables in one function by defining expressions of the second function. This is not surprising, because we can interpret the substitution of function variables as a deferred composition of the callee summary. The function variable expression serves as a placeholder for an evaluation function in an unknown callee until this callee is integrated by function variable substitution. A function variable expression is induced into the summary function computation by the instruction-level summary function of a call instruction. This summary functions model the effects of the call by function variable expressions and they do not integrate the callee immediately. This yields a open summary function representation where the composition of callee summaries still can be resolved later by the substitution of the function variables.

The observation is captured by the following lemma:

Lemma 4 (Correspondence of Function Substitution and Function Composition) *Let $\psi_i = \langle e_i^x \rangle$, and $\psi_{call_m} \circ \psi_i = (sm(e_i^x))$ be the composition with the instruction-level summary which uses the function variable expressions to defer the composition of the callee summary $\psi_m = \langle e_m^x \rangle$. Then,*

$$[\psi_{callm} \circ \psi_i]|_{[sm:=e_m^x]} = \psi_m \circ \psi_i$$

Proof 11 *Immediate consequence of the definition of function composition (Definition 12) and the definition of function variable substitution (Definition 17).*

Thus, the immediate composition of a callee summary and the composition with an open function representation that encode the effects of the call by function variables and the substitution of these function variables in a subsequent step yields the same result.

The substitution of function variables by applicable evaluation functions of callees establishes the bridge between open summary functions and applicable summary functions. Lemma 4 immediately reduces the properties of open summary functions to the properties of applicable functions:

Theorem 9 (Partial Order of Open Summary Functions) *The order of open summary functions is a partial order provided that all function variables in a open summary functions are substituted by applicable evaluation functions.*

Proof 12 *Consequence of the partial order of applicable summary functions (Theorem 6) and the correspondence of function substitution and function composition (Lemma 4).*

Theorem 10 (Monotony of Open Summary Functions) *Open summary functions are monotone in* $(L, \sqcap_L)$ *provided that all function variables are substituted by applicable evaluation functions.*

Proof 13 *Consequence of the monotony of applicable summary function (Theorem 8) and the correspondence of function substitution and function composition (Lemma 4).*

All in all, open summary functions form a lattice with respect to the meet operation of summary functions like applicable summary functions do. Therefore, the computation of open representations of intraprocedural summary functions is a data flow problem and the validator can check open summary function representations according to the general validation principle.

Additionally, the correspondence between function variable substitution and function composition also ensures that the substitution of an evaluation function preserves the order relation in the following sense:

Theorem 11 (Order Preservation by Function Substitution) *Let* ef_1, ef_2 *be defining expressions of two evaluation functions and* $e = s(ep)$ *be a function variable application expression. Then*

$$ef_1 \sqsubseteq ef_2 \qquad \Rightarrow \qquad e|_{[s:=ef_1]} \sqsubseteq_{E\downarrow} e|_{[s:=ef_2]}$$

Thus, if two expressions are in order relation, then the substitution of a function variable by this evaluation functions yields result functions with are in order relation with respect to the order of expressions. As a consequence, the first expression always evaluates to at least as conservative results than the second.

Proof 14 *Immediate consequence of the definition of function variable substitution (Definition 17) and the fact that function composition preserves the order of defining expressions (Theorem 7).*

This final result justifies the validity of the stepwise substitution of callee summaries into open representations of summary functions during an incremental validation process.

To summarise, the introduction of function variables provides a mechanism to compute open summary functions that express the effects of code which is external with respect to the software module under consideration. Additionally, definition of a substitution of function variables with callee summaries allows for a deferred integration of callee summaries as they become available. The

computation of the open summary function representations is a data flow problem so that open representations can be validated according to the general validation principle. Furthermore, the integration of callee summaries into an open function representation emulates the direct integration of callee summaries during the analysis phase. Therefore, the substitution of function variables is a mechanism to subsequently integrate callee summaries which become available during the validation process.

5.4.6 Function Variables in the Expression Model

Function variable application expressions fit smoothly into the model of data flow expressions because they can be treated like elementary function application expressions. The central challenge is to extend the normalisation process and to redefine the structural check of the order relation for open expressions.

Extension of the Normalisation Process

The constant folding, the duplicate variable removal, and the bottom shortcut reduction are not affected by the introduction of function variable expressions.

In contrast, the push out upper bound normalisation has to consider function variables. It does not only have to optimistically approximate data flow variables but function variables, too. This is achieved by the substitution of all function variable expressions with an optimistic upper bound for the analysis in question. Thus:

$$\begin{aligned} \text{If} \quad & [t(p)]|_{[x_i:=\top, s_i((ep^x)):=\top]} \sqcap c_{old} = c_{new} \sqsubset c_{old} \\ \text{then} \quad & t(p) \sqcap c_{old} \overset{POUB}{\longrightarrow} t(p) \sqcap c_{new} \end{aligned}$$

This way, the $\overset{POUB}{\longrightarrow}$-normalisation can be applied to elementary transfer function expressions even in the presence of function variables in the parameter expressions.

Additionally, we extend the distributivity normalisation to function variable application expressions:

$$\begin{aligned} t_i(p_1) \sqcap_L t_i(p_2) &\overset{DSTR}{\longrightarrow} t_i(p_1 \sqcap_L p_2) \\ s_i(p_1) \sqcap_L s_i(p_2) &\overset{DSTR}{\longrightarrow} s_i(p_1 \sqcap_L p_2) \end{aligned}$$

This ensures, that there remains at most one expression for each function variable on each nesting level of the data flow expression.

The extensions preserve the properties shown in Section 5.3.1. However, the extension of the distributivity rule requires that the evaluation functions

which are substituted for the function variables are distributive. The following theorem states that applicable summary functions are distributive provided that elementary transfer functions are distributive:

Theorem 12 (Distributivity of Applicable Summary Functions) *Applicable summary functions are distributive with respect to $\sqcap_L$. Let $\psi \in \Psi_{app}$:*

$$\forall v, w \in L : \psi(v) \sqcap_L \psi(w) = \psi(v \sqcap_L w)$$

Proof 15 *Let $e \in E_{app}$ be the defining expression of ψ. According to the definition of function application the proposition reduces to:*

$$\forall v, w \in L : e|_{[x:=v]} \sqcap_L e|_{[x:=w]} = e|_{[x:=v \sqcap_L w]}$$

By induction over the structure of applicable expressions:

$$\begin{array}{rcl}
e = \bot & : & \bot \sqcap_L \bot = \bot \\
e = c & : & c \sqcap_L c = c \\
e = x & : & v \sqcap_L w = v \sqcap_L w \\
e = e_s \sqcap x & : & e_s|_{[x:=v]} \sqcap_L v \sqcap_L e_s|_{[x:=w]} \sqcap_L w = e_s|_{[x:=v \sqcap_L w]} \sqcap_L v \sqcap_L w \\
e = t(e_s) & : & t(e_s|_{[x:=v]} \sqcap_L t(e_s|_{[x:=w]}) = t(e_s|_{[x:=v \sqcap_L w]})
\end{array}$$

Where the two last cases require the induction hypothesis that $e_s|_{[x:=v]} \sqcap_L e_s|_{[x:=w]} = e_s|_{[x:=v \sqcap_L w]}$ for all expressions e_s with a smaller maximum nesting depth than e. Furthermore, the last case requires that $t \in T$ is distributive.

Thus, the normalisation process can be extended in a straight-forward way. The normal form of an extended data flow expression is still unique and the comparison criterion for data flow expressions is still simple to check because the normal form has the following structure:

$$\begin{array}{rl}
 & e\downarrow = \bigsqcap_{i \in TI} t_i(p_i) \sqcap_L \bigsqcap_{j \in SJ} s_j(q_j) \sqcap_L \bigsqcap_{k \in VK} x_k \sqcap_L c \\
\text{or} & e\downarrow = \bot
\end{array}$$

Therefore, the validator can check open summary functions by the same means as their applicable counterparts. This completes the integration of open summary functions into the summary function model and enables the incremental validation of analysis results of modular software.

Remark: Nesting Depth Revisited

In the discussion of the expression model in Section 5.3.2 we already observed that elementary function application expressions lead to nested data flow expressions and that we have to restrict the nesting depth in order to keep the size of the expression representation under control. Similarly, function variable application expressions lead to nested data flow expressions as well. We can apply the same safe but conservative strategy to restrict the nesting depth in the presence of nested function variable expressions.

Both approximations break cyclic dependencies which stem from loops in the control flow. However, they conservatively approximate different dependencies. The limitation of the nesting depth of elementary transfer functions deals with situations where the result value of an elementary transfer function is used as a parameter for the same function application in the next iteration. Thus, the potential loss of precision depends on the properties of the elementary transfer functions. Additionally, the properties of elementary transfer functions can justify more precise approximation mechanisms as outlined in Section 5.2.2.

In contrast, the limitation of variable application expressions deals with situations where the result value of an external call is used as an argument of another call. If a parameter of a call depends on the result of the same call of a previous iteration of a loop in the flow graph, then the limitation of the nesting depth breaks a cyclic dependency pessimistically, which could have been resolved by a fix-point iteration in the interprocedural summary function computation phase.

Furthermore, a limitation of the nesting depth can even affect open summary function representations of straight line code. An early composition of subsequent summary functions which contain a reference to an external call leads to a safe approximation if a single data flow fact transitively depends on several external calls. The separation of the program state into an environment, the safe approximation at join points, and a limited lifetime of data flow facts restrict the number of situations where the limitation of the nesting depth reduces the precision of the analysis result. However, from a conceptual point of view the composition of open summary functions limits the number of external calls on a program path and approximates the effects of the preceeding path by a safe lower bound.

Additionally, the practical experiences with the current prototype implementation of the framework reveal that the early substitution of parameter expressions has a significant impact on the runtime requirements of the analysis phase. We discuss the problem and a potential solution as part of the runtime comparison of the analysis and the validation phase in Section 9.4, but stick with the old formulation of the function variable model the current prototype implementation is based upon throughout the thesis.

5.5 Method Invocation and Parameter Passing

Section 4.2.4 already provides an overview of the semantics of method invocation in the presence of local variables and parameter passing. The general observation is that the call site model requires two additional functions. The summary function ψ_{call_m} models the assignment of arguments to parameters and the return functional ψ_{ret} acts as a selector function which maps modifications back into the context of the caller and restores the unaffected rest of the context of the caller.

This section defines an appropriate representation for the program state in the summary function model and defines the required summary functions.

5.5.1 Local Variables, Parameters, and Global Variables

The summary function representation models the program state as a environment which maps an arbitrary set data flow variables to data flow values. We have not fixed the semantics of data flow variables, so that different analyses can use them in different ways. For example, copy constant propagation uses data flow variables to represent program variables directly, while an available expression analysis models expressions in the program by data flow variables. Many intraprocedural analyses consider the data flow through local variables. If such analyses are extended to the interprocedural case, the data flow through parameters, return values and global variables has to be considered, too. The straight-forward way is to represent these different kinds of variables directly by data flow variables in the data flow tuple.

We assume without loss of generality that local variables, parameters, and global variables can be identified by a unique number up to an upper bound λ, π, and γ, respectively. The following definition combines all of these variables into a tuple which serves as a model for the program state.

Definition 18 (Interprocedural Core Tuple) *Let* $LV = \{l_1, \ldots, l_\lambda\}, P = \{p_1, \ldots, p_\pi\}$, *and* $GV = \{g_1, \ldots, g_\gamma\}$ *denote the sets of local variables, parameters and global variables respectively. Then the set of data flow variables is defined as* $Var = LV \cup P \cup GV \cup r$ *and the* interprocedural core state *consists of the tuple*

$$(l_1, \ldots, l_\lambda, r, p_1, \ldots, p_\pi, g_1, \ldots, g_\gamma)$$

where r is a special variable which represents the result value of a method invocation.

Some comments are advisable. The sets LV and P represent the local variables and parameters of a method invocation. However, it is not necessary to model the local variables and parameters of each method separately, because the variables can be reused for each method as we will see in the subsequent sections. Thus, the sets are limited by the maximum number of local variables and parameters which occur in some method of the specific program. We model parameters explicitly because from the perspective of program analysis

they share properties with local variables and global variables: on the one hand data is passed to a callee via parameters and global variables while on the other hand each method has an own set of parameters like it has an own set of local variables. This differs from global variables which are unique throughout the whole program.

Both properties are relevant for the model of method invocations.

Remark: Extensions of the Program State The definition of the interprocedural core tuple captures the data flow within the procedural core of a programming language. It is possible to analyse the data flow through the call stack of the program because the stack is modelled by local variables and parameters.

Furthermore, it is possible to analyse the data flow through uniquely defined variables like global variables. Global variables correspond to class attributes in object-oriented languages. Unlike the number of local variables and parameters - which is usually very limited - the number of global variables can be linear in the size of the program. However, special encoding strategies can reduce the size of the program state representation as outlined in Sections 8.2.4 and 6.3.1.

The extension to objects and their fields complicates the issue. A simple but limited approach is to extend the program state tuple by a single representative for each object field. [3]. However, the increase of precision which can be gained by this extension may very well be limited because the analysis can only compute data flow information which is valid for all object instances.

A points-to analysis is required whenever an analysis tries to restrict the potential objects which are accessed at a specific program point which reads or writes an instance field. Given that valid results of a point-to analysis are at hand, a subsequent analysis phase can restrict the potential effects of a field access to fields of the object instances which may be referenced at the access site. This requires an extended representation of the program state. The usual way is to use a different representative for a fields for each object instantiation site within the program. Obviously, this increases the number of field representatives significantly and special optimisation strategies may very well be required to keep the size of the state representation under control.

We intentionally limit the discussion to the procedural core model in order to highlight the fundamental principles of the validation of analysis results. The extension of these principles to more sophisticated analyses should always be attempted in the straight-forward manner outlined in this section and technical challenges like the size of the tuple representation can be approached by technical means. However, the summary function model may still degenerate for more sophisticated analyses, especially if pointer analyses are considered. This question is an interesting direction of further research.

[3] We describe the situation for the programming language Java, where at least the type of the object and the specific field is known at each field access site. In languages like C which allow for arbitrary pointer arithmetic the simple approach does not work because virtually any field may be affected when a value is written to a storage location identified by a pointer

Nevertheless, the core model of summary function analysis is already capable to deal with interesting program analysis like the type inference analysis which is described in Section 7.3. The result of this analysis yields an interprocedural call graph which is a prerequisite for any interprocedural analysis.

5.5.2 Parameter Passing and the Call-Function

The runtime environment creates a new activation record[4] on the call stack of the program whenever a method is called. The activation record contains local variables and parameters so that each method invocation operates on its own set of local values. In contrast, each method invocation can access global variables uniformly. The state of local variables, parameters, and global variables immediately before the execution of the first instruction of a method constitutes the invocation context of the method.

Interprocedural summary functions map the invocation context of the method to the program state immediately after the execution of the method. Thus, summary functions describe the manipulation of both the local variables of the current method *and* the manipulation of global variables.

The invocation context depends on the program state at a specific call site as depicted in Figure 5.13.

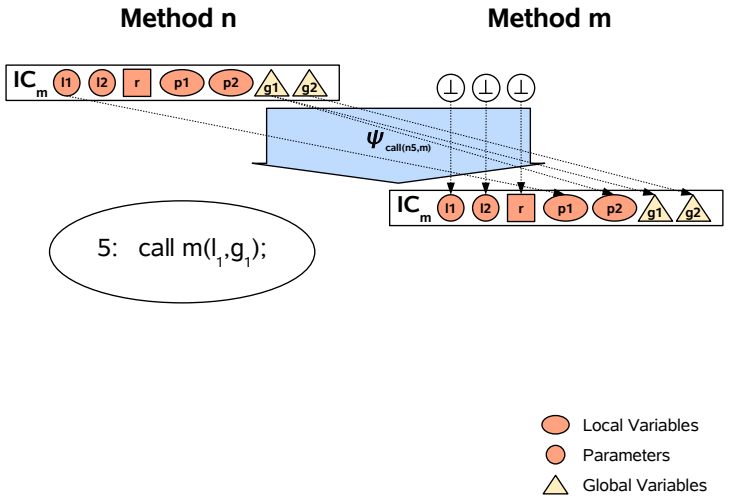


Figure 5.13: Construction of the Call Function

The arguments of the method call initialise the parameters of the callee. Any kind of the variables - like local variable l_1 and global variable g_1 - can serve as an argument. Furthermore, the values of global variables coincide. In contrast, local variables of the callee do not depend on the program state at the call site

[4] An activation record is called "method frame" in the Java terminology.

because they are initialised to default values according to the semantics of the programming language in question.

A program analysis describes these dependencies between the representation of the program state at the call site and the invocation context of the callee by a "call"-function. The call function depends on the call site, because the call site determines the arguments of the call. Furthermore, the call function depends on the callee because it maps the arguments to appropriate parameters and initialises the local variables. Therefore, we subscript each call-function with the program point of the call in the caller and with the name of the callee.

The transfer of values from arguments to parameters is closely related to a sequence of assignments. However, these assignments have to happen *simultaneously* in order to avoid interference between the local variables and parameters of the caller and the callee. Consider the call instruction `call(p2, p1)`. Obviously, the assignment sequence

```
p1 = p2;
p2 = p1;
```

does *not* produce the correct invocation context, because the *updated* value of $p1$ which corresponds to the parameter of the callee is used to initialise parameter $p2$. Fortunately, the summary function representation is able to express simultaneous updates of variables directly in the following way:

Definition 19 (Call Function) *Let* $call_{(n_x,m)}(v_1,\ldots,v_\phi)$ *be an invoke instruction which calls method m at point x in method n. Then, the* call-function $\psi_{call(n_x,m)}$ *is defined as:*

$$\psi_{call(m_x,n)}(l_1,\ldots,l_\lambda,r,p_1,\ldots,p_\phi,g_1,\ldots,g_\gamma) =_{df} \langle \perp_1,\ldots,\perp_\lambda,\perp,v_1,\ldots,v_\phi,g_1,\ldots,g_\gamma\rangle$$

5.5.3 Method Return

The call-function maps the program state at the call site to the invocation context of the callee. The interprocedural summary function of the callee maps this invocation context to the program state immediately after the execution of the method. However, the output state of the interprocedural summary function expresses the program state *in terms of the callee*. Especially, this program state contains information about the activation record of the callee and not about the invocation context of the caller.

However, the summary function which captures the semantics of the method invocation within the caller is a program state transformer which manipulates the activation record of the caller. Thus, the program analysis model has to integrate potential modifications into the context of the caller: Firstly, the manipulations of global variables become visible in the callee after the call. Secondly, the result value of the method invocation is stored to a variable in the caller context.

Furthermore, local variables and parameters of the caller are *not* effected by the method invocation. Thus, the original values have to be restored upon the method return. The situation is depicted in Figure 5.14.

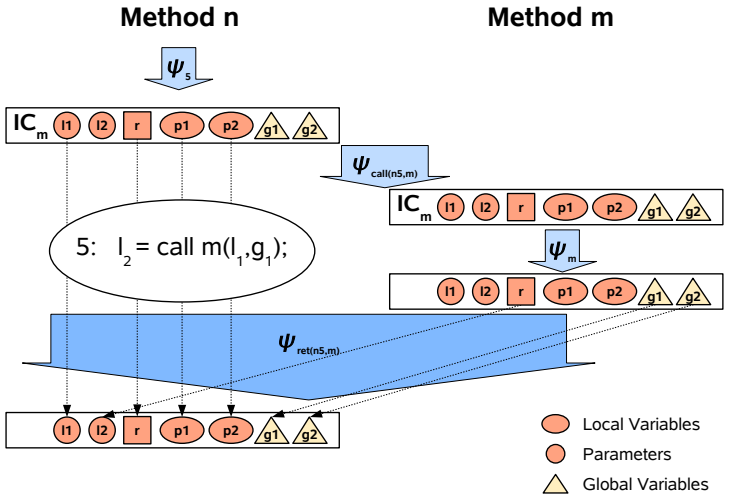


Figure 5.14: Return Function

The return-function restores the values l_1, r, p_1, and p_2 in the caller context because they are not affected by the call. Furthermore, the return-function transfers the values of the global variables from the callee context into the context of the caller, because the manipulation of global variables during the method invocation affects invocation context of the caller. Finally, the result value is stored into local variable l_2 according to the assignment statement in node 5 of the caller. This effect occurs *after* the restore of the local and the transfer of global variables.

Formally, the return-function can be expressed as a functional which takes two summary functions as input and produces a result function for the call instruction. The first summary function is the intraprocedural summary ψ_5 encodes the program state in the caller immediately before the execution of the method call. The second summary function is the function composition $\psi_m \circ \psi_{call(n_5,m)} \circ \psi_5$ which describes the program state immediately after the execution of the callee. The functional $\psi_{ret(n_5,m)}$ yields a function which either uses the mapping in the first or the second summary function to determine its result - depending on the kind of variable in question.

Definition 20 (Return Function) *Let* $v_r = call_{(n_x,m)}(v_1, \ldots, v_\phi)$ *be an invoke instruction which calls method* m *at point* x *in method* n. *Then the* return-function

$\psi_{ret(n_x,m)}$ is defined as:

$$\psi_{ret(n_x,m)}(\psi_I, \psi_c) = \psi_r \circ \psi_{select} with$$
$$\psi_{select}(l_1, \ldots, l_\lambda, r, p_1, \ldots, p_\pi, g_1, \ldots, g_\gamma) =_{df} (e_I^{l1}, \ldots, e_I^{l\lambda}, e_I^r, e_I^p, \ldots, e_I^{p\pi}, e_c^{g1}, \ldots, e_c^{g\gamma})$$
$$\psi_r(l_1, \ldots, l_\lambda, r, p_1, \ldots, p_\pi, g_1, \ldots, g_\gamma) =_{df} (id_e, \ldots, id_e, v_r = e_c^r, id_e \ldots id_e)$$

where ψ_r uses the defining expression of the result value in the result representation in the "appropriate" place v_r and maps all other variables to themselves (id_e).

All in all, the semantics of a call instruction can now be expressed as follows:

Definition 21 (Summary Function of a Call Instruction) *Let $v_r = call_{(n_x,m)}(v_1, \ldots, v_\phi)$ be an invoke instruction which calls method m at point x in method n. Then the instruction-level summary function of the call instruction ψ_c is defined as*

$$\psi_c(s) = \psi_{ret(nx,m)}(s, \psi_m \circ \psi_{call(nx,m)} \circ s)$$

Remark The model assumes that the data flow information in local variables cannot be affected by a method invocation which is true for all analyses described in Chapter 7. Programming languages like C which allow a direct manipulation of the call stack or sophisticated points-to analyses can complicate the issue. However, the simplifying assumption which holds for the considered analyses allows the reuse of local variables and parameters and avoids a model for the whole call stack in the analysis.

Furthermore, the representation implicitly models call-by-value semantics and does not deal with aliasing effects. Once again, this is sufficient to deal with the analyses presented in this thesis. An extension of the framework which takes aliasing effects into account requires a validatable variant of an alias analysis. The question which alias analysis are expressible in the validatable summary function model is an interesting direction of further research.

5.5.4 Properties of Call- and Return-Function

Both the call- and the return-function have to be summary functions, in order to integrate the extended model of a method invocation smoothly into the summary function framework.

This is obviously the case for call-functions. Consequently, the function composition $\psi_m \circ \psi_{call} \circ \psi_I$ which yields the program state after execution of m is also a summary function, because the composition of two summary function again yields a summary function.

The return-function also constructs a summary function because it just point-wise selects the evaluation functions of the given functions. This point-wise selection is guided by the kind of the variables - evaluation functions for local variables are taken from the caller context and the evaluation functions of the

global variables and the result value are taken from the callee summary. This ensures, that the instruction-level summary function of a call is monotone with respect to function composition. This is important to ensure, that the instruction-level summary is a valid transfer function for the call instruction.

Consequently, the extended model for method invocation fits directly into the definition of the data flow problem which computes interprocedural. Thus, the validation process can be immediately applied for the extended model, too.

5.5.5 Related Approaches

The formalisation of method invocation instruction presented in this section is an adoption of the call-site model of the interprocedural framework of Knoop [Kno99] to the summary function representation based on data flow expressions. The difference is that the original model defines summary functions as transformers of an abstract representation of the whole call stack. This model allows to specify the interprocedural meet over all path solution but requires the representation of a potentially infinite abstract call stack. In order to deal with this issue, the original framework specifies an algorithm which computes the interprocedural maximum fix point solution. This algorithm considers only two elements of the call stack: the activation record of the caller and the one of the callee.

The return-function defined in Section 5.5.3 follows this intuition. It also considers the topmost elements of the call stack only. The program state of the caller is encoded in the input summary function ψ_5 while the program state of the callee is encoded in the result of the function composition of input summary function, call-function, and the summary of the callee. The return-function merges the two states immediately after the method invocation has finished.

Such an *early merge* is the key difference between the meet over all path solution and the root cause for the loss of precision for non-distributive problems. The phenomenon is usually observed when two intraprocedural paths join after a conditional or after a loop. Here, we observe the interprocedural counterpart, because the interprocedural summary function ψ_m already incorporates all potential call sequences which originate from the call. The return-function integrates this conservative approximation of the semantics of the call into the summary of the callee. It is enriched by additional early intraprocedural merges during the analysis of the caller. This yields a conservative approximation of the interprocedural summary which in turn affects the precision of each callee.

The interprocedural framework of Reps, Horwitz, and Sagiv [RHS95], [SRH96] integrates the semantics of parameter passing to and returning from a method invocation explicitly within the path compression algorithm. This involves three different sources of information. The compressed callee summary captures the semantics of the callee, a path grammar restricts the data flow to interprocedurally realisable paths, and an additional flow edge, which directly connects the call and the return nodes in the caller contributes local data flow.

The data flow expression model does not involve a path grammar - which is the fundamental modelling technique for the call-string approach also outlined by Sharir and Pnuelli [SP81] - because the functional approach directly inserts the summary function of the callee with respect to both, the call- *and* the return semantics. The additional flow edge within the caller in the model of Reps seems to be closely related to the fact that the result-function takes the input summary ψ_5 as a parameter and uses this summary to restore the local variables and parameters of the caller.

5.6 Summary and Comparison

The summary function model solves several important issues which are vital to support the validation of interprocedural analysis results of software modules:

- The definition of function composition, function meet, and function comparison is required to check the validity of summary functions according to the general validation principle.
- Summary functions have to be applicable because they are used as transfer function during the validation of the value computation phase of interprocedural analysis.
- The integration of arbitrary elementary transfer functions increases the expressiveness of the model and provides a mechanism to extend intraprocedural analysis to their interprocedural counterpart in a generic way.
- Analysis results for a single software module can be expressed by open summary functions which contain function variables that refer to the summary functions of other modules. This allows for both the determination of a safe lower bound of the available results as well as the subsequent composition of analysis results.
- The call-site model takes parameter passing and the influence of local variables into account. Furthermore, it provides a potential extension point for reference semantics.

The main focus of the model is to support the validation of analysis results of software modules. Efficiency issues - though addressed - are a secondary target. The generic formulation of the data flow expression model allows to deal with the central issues of the validation scenario in a way which is independent from a specific analysis. However, it does not utilise problem-specific knowledge like the compression of elementary transfer functions of the linear constant propagation analysis.

The following sections flesh out the capabilities and limitations of the summary function model by a comparison to related approaches.

5.6.1 Capabilities of the Summary Function Model

Expressiveness The expressiveness of the summary function model is closely related to the summary function model of Reps, Horwitz, and Sagiv [RHS95], [SRH96] as outlined in Sections 5.1.4 and 5.2.3. The models coincide for simple bit-vector analyses but differ in the integration of problem specific transfer functions. The graph model requires that the dependencies between elements of the program state can be decomposed into dependencies between single variables and a subsequent conservative approximation. In contrast, elementary transfer functions allow dependencies which involve more than a single input variable. Such dependencies cannot be integrated into the graph model smoothly, because they require the introduction of multi-edges.

All in all, the expression-based summary function model is capable to deal with an extended class of interprocedural distributive environment problems.

Recent efforts to compute generic summary functions for a wider class of problems include the "conditional micro transformer" approach of Yorsh [YYC08] and the "generic assertions" approach of Gulwani [GT07].

Conditional micro transformers express a summary function in terms of disjoined micro transformers which capture the transfer semantics only for a subset of all program states that satisfy an associated condition. Function composition involves the computation of weakest preconditions which in turn requires that micro transformers are invertible. The model can cope with IDE-problems but it is unclear to which class of analysis problems the approach extends. The approach is concerned with the simplification of compositional micro-transformers which seems to be related to the normalisation step in the data flow expression model. However, the simplification of conditional micro transformers does not explicitly address the challenge to keep the simplified form unique which is vital for an efficient validation of the representation.

The generic assertions approach extend the expressiveness of method summaries to program analyses which involve linear arithmetic [MOS04] and unary uninterpreted functions [MORS05]. Possible assertions contain equalities of expressions and require that the underlying theory is unitary, i.e. for all equalities there exists a unifier which is more general than any other unifier for that equality. This condition ensures the compactness of the representation of assertions and leads to a fast computation of fix-points in the presence of cyclic structures of the program. This aspect of the approach seems to be related to the normalisation rules of data flow expressions which ensure the uniqueness of the normal form of summary functions (refer to Section 5.3.1). The question whether a validation pass can check the representation of summary functions in terms of assertions efficiently may be an interesting direction of further research.

Call-Site Model The specification of the method invocation semantics in an interprocedural analysis framework can be discussed on two different levels. Firstly, the parameter passing mechanism and the integration of modifications

into the context of a caller immediately after the call site have to be expressed in terms of the summary function model of the framework. Secondly, the general mechanism has to be instantiated either by some kind of default implementation or specific to the data flow analysis in question.

All interprocedural frameworks have to deal with two different issues at call sites. An additional call-function has to capture the assignment of arguments to parameters before the summary of the callee is considered while a return-function has to integrate the effect of the call into the context of the caller. The return-function needs access to the program state immediately after execution of the callee and to the invocation context of the call because the values of unaffected local variables of the callee have to be restored.

The interprocedural framework of Knoop [Kno99] models parameter passing explicitly by the integration of an appropriate simultaneous assignment statement before each call instruction. Thus, the transfer function of assignment statements of the inducing data flow problem can be used directly. The framework extends the program state model to a abstract variant of the call stack in order to separate local variables of different method invocations. Summary functions operate on this stack representation. The instruction level summary functions just manipulate the topmost element and the call function pushes a new abstract instance of a method frame onto the stack. Consequently, the return function has access to an abstraction of the whole stack and can merge the two topmost elements after a call site.

The PAG framework [AM95] uses the call-string approach which does not explicitly compute function summaries but restricts the propagation of data flow values on interprocedurally realisable paths. To do so, the program state model holds information about different calling sequences and the program state after a method invocation is merged into "fitting" sequences only. This is achieved by user-defined "mapping"-functions, which have the information about different calling sequences available.

The graph reachability approach of Reps [RHS95] uses additional flow edges from call- to return-nodes to support restoring of local variables of the callee. These edges host an own data flow function which can immediately map parts of the invocation context to the result context in the caller.

PAG as well as the graph reachability approach capture the parameter passing mechanism by additional summary functions which augment the flow edge from a call-node to the entry node of the callee.

The method invocation mechanism presented in Section 5.5 solves the challenge in a similar way. The call-function is a summary function which expresses the simultaneous assignment of arguments to parameters. The return-functional takes two summary functions as input which describe the original invocation context and the situation after the execution of the callee respectively. Thus, the original state can be restored and modifications can be integrated into the result context of the caller.

Section 5.5 already defines a default implementation which is suitable to deal with the all analyses problems presented in Chapter 7. This model assumes that the local variables of the caller cannot be affected by the callee. This is true for several analyses especially when Java programs are considered because the Java runtime environment prevents a direct manipulation of the call stack. Nevertheless, the call- and return-function can be replaced by more sophisticated versions if this is required for additional analyses.

All in all, interprocedural analysis frameworks tackle the same problems at call sites even though the abstractions differ significantly. The data-flow expression based summary model essentially proceeds along the same lines. The most important advantage of our formalisation is that it keeps the representation of summary functions validatable. This is an aspect that has not been considered from the perspective of a interprocedural framework yet.

Modular Analysis Traditional analyses usually expect the whole program to be present at analysis time or make worst-case assumptions about invocations of unknown methods.

The analysis of software modules requires a result representation which can be subsequently composed to the final result for the whole program. Such a result representation can either be tailored to a specific problem or an analysis framework tries to deal with the issue in a generic way.

The first approach can additionally be divided into two subcategories: either an analysis produces problem specific summary functions or it uses a completely problem specific representation. Examples of the first kind of analyses include the points-to summaries of [RR01] or [GR07]. The advantage of specialised summary functions functions is that their composition can be expressed in terms of the functional approach to interprocedural analysis. However, the representation of call-backs - if permitted - requires the integration of functional aspects into the problem specific result representation. The second kind of analyses which provide a problem specific solution define both the result representation and the composition mechanism in an own model. An example is the compositional pointer and escape analysis specified in [WR99] where a so-called points-to escape graph encodes the relationships of references and the composition of results is reduced to a specialised graph union.

In contrast to the problem specific approaches, the composition of partial analysis results can also be tackled based on the abstractions of an interprocedural framework. The component-level analysis of Rountev [Rou02], [RKM06] tries to perform as much of the summary function computation phase as possible. The key observation is that it is possible to compute the summary functions of leaf methods independently from the rest of the program because they do not call any method. Furthermore, the summary functions of leaf methods can already be inserted into the summary function computation of their callers which in turn can make additional summaries computable. The computation stops whenever the call to an external method is encountered. The result is

a compressed representation of the intraprocedural summary functions of the software module. The composition of the results just corresponds to the continuation of the summary function computation with newly available summary functions. Partial analysis systems defined by Thies [Thi02] constitute another approach to specify analysis results of a software modules in a generic way. Partial analysis systems encode data flow results in an algebraic structure which is closely related to the structure of a data flow problem but additionally offers the possibility to express dependencies between data flow facts. So-called single aspect models establish references to data flow facts from unknown software modules within the algebraic model. The approach can automatically combine analysis results of different modules at link-time - given that the results of a specific analysis can be expressed in terms of a partial analysis system,.

The summary function representation defined in this thesis is also a generic approach to the combination of modular results. The incremental computation and validation of open summary functions follows the general idea of the component-level analysis. However, function variables which express dependencies on external modules are already integrated as first class elements in the model. In particular, the normalisation process can affect function variable expressions. This way, the analysis of a single software module drops dependencies on other software modules automatically if they cannot influence the result anymore. This is more aggressive than the component-level analysis which stops the computation process of summary function as soon as a dependency on an external summary is encountered.

Furthermore, the validatable variant of a type inference algorithm which is discussed in Section 7.3 solves an important challenge more accurately than other approaches: the result can detect situations where a dynamically bound method call targets methods in the software module only. Thus, even dynamically bound calls can be closed already during the analysis of the software module.

Data flow expressions and the normalisation specified in Section 5.3 share several ideas with the algebraic transformations in partial analysis systems. Additionally, data flow expressions provide several improvements:

- The dependency on the inducing data flow problem is made explicit in data flow expressions. This eases the specification of data flow results in the expression model. Furthermore, the normalisation rules are solely based on generic properties of the inducing data flow problem.
- Method invocations are integrated directly into data flow expressions. In contrast, single aspect models for methods rely on the additional concept of method families which is specified separately.
- The reduction rules for data flow expression ensure the uniqueness of the normal form which is a central prerequisite for the validation of data flow expressions

5.6.2 Limitations of the Summary Function Model

Influence of Elementary Transfer Functions The data flow expression model targets both the validation aspect and the representation of modular analysis results without taking advantage of special properties of a specific analysis. This way, we can discuss and solve the main challenges of our application scenario in a uniform way which keeps the focus on the general validation principles. The approach can deal with an interesting class of analysis problems. Particularly, the model is able to express a data-flow based type inference algorithm that is useful in the incremental validation scenario. The type inference result yields a call graph which in turn is a prerequisite for other interprocedural analyses.

However, the universal formulation comes at a cost. The key question is whether the use of elementary transfer functions and function variable application expressions can be kept under control. Elementary transfer functions abstract from the problem specific details but lead to nested expressions. The nesting depth can be infinite if the result of an elementary transfer function is used as a parameter of the same transfer function. This can occur in intraprocedural contexts whenever a data flow value computed in a preceding loop iteration contributes to the same computation in the subsequent loop iteration. The nesting depth is closely related to the questions how much subsequent applications of transfer function lead to a fix-point. This is an important *problem specific* property the data flow expression model is not aware of. The restriction of the maximum nesting depth leads to a loss of precision while deeply nested expressions can only be omitted if the analysis exploits knowledge about the function properties of the specific problem.

A second consequence of the way the model deals with elementary transfer function is that nested expressions cannot be compressed in a problem specific way. For example, linear functions represent the symbolic computations of the linear constant propagation problem. The composition of two linear functions can be compressed into a single linear function because

$$lin_{(a1,b1)}(lin_{(a2,b2)}(x)) = a_1(a_2x + b_2) + b_1 = (a_1a_2)x + (a_1a_2 + b_2) = lin_{(c,d)}(x)$$

The extension of the graph reachability approach to linear constant propagation [SRH96] explicitly exploits this compression strategy to keep the size of the micro transformers under control. This is not immediately possible in the data flow expression model because it has to be extended to integrate such problem specific compression techniques.

Nevertheless, three mechanisms counter the blow up of elementary transfer functions in the expression model. Firstly, the generation of data flow facts removes any complex expression that describes the previous state of the data flow fact in question. Secondly, the conservative approximation with a lower bound drops a potentially complex expression. This occurs whenever a path

where the pessimistic assumptions about a single data flow fact have to be made is joined with other paths. Thirdly, the $\overset{POUB}{\longrightarrow}$-normalisation makes an upper bound for elementary function application expressions available on the level of the the function application. This removes elementary transfer functions that cannot contribute valuable information anymore.

To summarise, the model of data flow expressions is especially well suited to express analyses problems, which

- do only require few elementary transfer functions
- have many generation points of data flow information
- lead to a significant amount of safe lower bounds either because they safely approximate the influence of data flow which is not considered by the analysis or because they do not always yield valuable information.

Interestingly, bit-vector problems do usually not require elementary transfer functions at all - so that the data flow expression model stays nearly as efficient than problem specific solutions. In the worst case, the summary function model can degenerate to an explicit representation of the composition and meet of summary functions induced by the control flow of the program. In this case, the tuple representation is not very efficient because it encodes the structure of summary functions for each data flow fact again.

Model of Program State The current description of the program state is tailored to the representation of a data flow fact per variable in the program. It is especially useful to analyse the data flow through local variables on the call stack of the program. The reason is that in many analysis the execution of a callee cannot influence the local variables of the caller except for the variable the result of the call is assigned to. As a consequence, the environment model effectively isolates the influence of a call because only one variable becomes dependent on a function variable expression.

This advantage is reduced when global variables and other program entities are integrated into the environment. The invocation of a method can influence any of these values so that each of them depends on a function variable expression after the call. Similarly to elementary transfer functions, conservative approximations and the generation of new data flow values can remove such references. Furthermore, the challenge can also be addressed on a technical level as discussed in Section 8.2.5.

Algebraic Properties of Summary Functions The summary function model requires that elementary transfer functions are distributive with respect to the safe approximation operator of the inducing lattice. Essentially, this property guarantees that the result of the evaluation of an expression does not depend on the order of the evaluation of its subexpressions. Therefore, the different normalisation steps can be applied in an arbitrary order without changing the

result of the expression. This keeps the validation phase flexible and simplifies the discussion about the formal properties of the normal form.

Interprocedural frameworks limit themselves to distributive analyses whenever guarantees about the quality of a solution have to be established. The influence of non-distributivity is subtle: the functional approach has to keep the summary function representation in the first phase under control, while the call-string approach has to limit the depth of the call sequence which is tracked precisely. The underlying question is always how long different program paths are modelled separately before they are joined. Any early join has the potential to loose precision for non-distributive problems.

We have identified two ways to approach the validation of non-distributive problems. Firstly, the most conservative result with respect to the loss of precision caused by early joints can be transmitted. This prevents that the validator can accidentally join information too early and end up with a result which is to weak to validate the given result. The second way is to synchronise the analysis and the validation phase so that the loss of precision is guaranteed to occur at the same points. This aspect is not investigated further.

Path Sensitivity The current representation of summary functions captures pure data flow only. It does not take dependencies between data flow values and the expressions of control statements into account. Therefore, the model is not able to deal with conditional data flow analyses.

Nevertheless, extensions like the path-sensitive reformulation of the graph reachability approach in the Bebob system [BR01] - which is part of the SLAM-project [BR02] - can also be possible in the data flow expression model. Such an extension leads to a more complex representation of the program state tuple because it has to be able to express that some data flow facts are valid only if conditions about other data flow facts hold. The Bebop system achieves this by reducing the program to a boolean program so that transfer functions become boolean functions which can be efficiently represented by binary decision diagrams. The reduction to a boolean program works well for analyses which determine "yes or no"-decisions. However, it is not obvious how this approach extends to arbitrary analyses problems. It is likely, that the same phenomenon can be observed when a path sensitive extension of data flow expressions is considered because the restricted class of boolean functions can yield additional reductions of the extended representation.

Relational Analyses Muchnick and Jones investigate the general complexity of flow analysis in [JM81]. They differentiate two classes of analysis methods - the *independent attribute method* and the *relational method*.

The algorithms that use the independent attribute method associate with each program point I a function $f_I : \{X_1, \ldots, X_n\} \rightarrow D$ where $X_1, \ldots X_n$ are the variables of the program and D is a lattice of data flow elements which describe properties

of a variable. For example, $f_I(X_k) = \{\text{bool}, \text{int}\}$ states that variable X_k may have type *bool* or type *int* at program point I in a type inference analysis.

A system of simultaneous equations of the form $f_{I_i}(X_k) = g_{i_j}(f_{I_1}(X_1), \ldots, f_{I_m}(X_n))$ specifies the problem and can be solved by fix-point iteration.

In contrast the relational method associates a relation $f_I \subseteq D^n$ with each program point with the interpretation that f_I is a *set* of n-tuples describing the relationships among the values of $X_1, \ldots, X_n$ at point I [JM81]. Assume that the analysis in question performs a type analysis on the example program depicted in Figure 5.15.

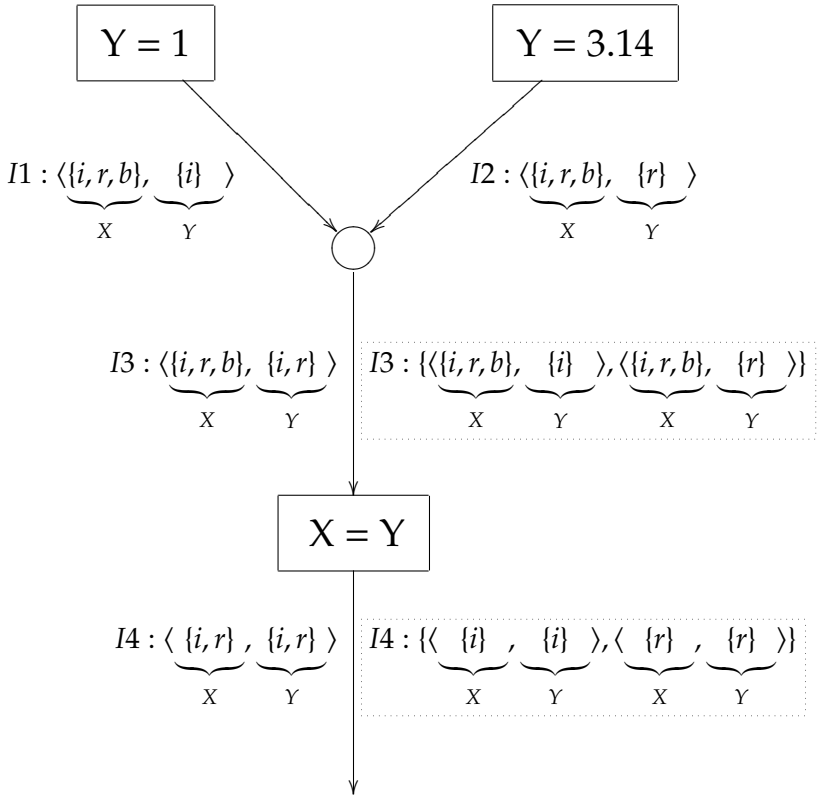


Figure 5.15: Comparison of the Attribute and the Relational Method

Two different paths in the program flow merge immediately before the node which contains the assignment `X = Y`. The variable *Y* has type `integer`[5] on the left path and type `real` on the right path.

The attribute method associates a single data flow value to each variable at each program point. Therefore, it merges the two different types that variable *Y* may have at the join point I_3. The result of this operation is the type set $\{i, r\}$ which indicates that *Y* has either type `integer` or `real`. As a consequence, the analysis infers that *X* also has either type `integer` or type `real` at program point I_4 after the assignment. This approach is not capable to detect the *relation* between the

[5] Abbreviated by *i*

type of X and the type of Y - the information that X and Y have the *same* type at point I_4 is lost. The reason is that the attribute method operates on a single representation of the program state at each program point. Therefore, it has to merge the different states from the left and the right path at the join point.

The relational approach solves the problem, because it uses a *set* of data flow tuples as depicted by the dashed-boxed values in Figure 5.15. The two tuples which represent the program state on the different paths are *not* merged at the join point but combined by *set union* into a new set which contains two different tuples for the different program states. A relational analysis considers the effects of a code block on all program states in isolation. This yields a new set of program states for I_4 which clearly states that X and Y have the same type at this point.

Obviously, the relational approach is a generalisation of the independent attribute approach, because the result of the attribute approach is always the conservative approximation of all tuples in the tuple set of the relational approach at the same point. Essentially, the relational approach has the capability to keep track of different program states from different execution paths. This increases the expressiveness of the approach. However, it increases the computational complexity, too. In [JM81] Muchnick and Jones show that the independent attribute approach is in $\mathcal{P}$ while type checking a language proposed by Dijkstra in [Dij76] with the relational approach is in $\mathcal{NP}$.

The summary function model for the validation of interprocedural analysis results follows the design principles of the attribute approach. A summary function transforms a *single* input state to a single output state. Internally, the program state is also decomposed into a tuple of properties of a set of data flow variables and a summary function is decomposed into a single evaluation function for each variable in the output state. This closely resembles the intuition of the system of equations of the form

$$f_{I_i}(X_k) = g_{i_j}(f_{I_1}(X_1), \ldots, f_{I_m}(X_n))$$

which also specifies the dependence of the flow value of X_k at the point I_i in terms of the data flow facts of other variables at some other program points. A summary function ψ_{ij} which maps the whole program state at point I_i to the program state at point I_j is defined as

$$\psi_{ij}(\langle X_1, \ldots, X_n \rangle) = \langle e_{ij}^{x1}(X_1, \ldots, X_n), \ldots, e_{ij}^{xn}(X_1, \ldots, X_n) \rangle$$

Thus, the evaluation functions correspond to the functions g_{i_j}. A difference is that a summary function always considers a specific program point I_i as the input point and refers to the state of the variables at this point only. In contrast, the definition of the equation system in the attribute approach can refer to different program states I_1 and I_k for example.

It is an interesting question whether the summary function model can be extended to the relational approach as well. An immediate idea is to use a

set of summary functions instead of a single summary function to express the mapping of program states from one point to another. Each of these summary functions can map one of the input states to the corresponding output state, so that the set of evaluation functions can act as a transformer for the extended program state model in the relational approach. The function meet is the most important point to consider because the behaviour at join points seems to be the key difference between the independent attribute approach and the relational approach. The first approach implies a conservative approximation of program states while the second one implies the set union of program state sets. Currently, the function meet is defined by a reduction to the conservative approximation operator of the inducing lattice which is another hint at the relationship to the attribute approach. The extension to relational analysis seems to be likely to require some kind of union operation on summary function sets. This introduces an additional level of abstraction into the summary function model and is beyond the scope of this thesis.

6 Optimisation of the Validation Process

The validation of data flow results is reasonable only if it is significantly more efficient than the iterative data flow algorithm. One of the key properties of validation is that it avoids iterative fix point computations. Therefore, a single pass over the system of data flow equations suffices to validate given results. However, the annotation of data flow results increases the size of the transmitted data. Furthermore, it may even be impossible to store the whole data flow result at the consumer site which is why it is beneficial to use at least parts of the result ahead of time and to drop data flow information as soon as it is no longer needed.

Efficiency concerns can either be approached by an improvement of the underlying algorithm or by technical improvements like the use of more efficient data structures. Algorithmic ideas exploit for example that the lifetime of data flow facts in the validation process is limited or that it is possible to reuse data which is computed during validation. In contrast, technical means include efficient encoding strategies for data flow elements or specialised data structures. This chapter discusses algorithmic improvements while the discussion of technical optimisations is postponed to Chapter 8.

The fundamental ideas discussed in this chapter can be summarised as follows:

Reduction of the Certificate The validator produces data flow facts during the checks required in the validation process. Parts of these values coincide with the final data flow result. Thus, only those pieces of data flow information which are not reconstructed in the validation process have to be transmitted in the certificate.

Lifetime of Data Flow Values Some data flow facts influence only a limited number of other data flow facts. Thus, a data flow fact can be dropped as soon as all dependent data flow facts have been validated.

Intentional Under-approximation The validation process is capable to validate *any* fix point of a given data flow problem. Therefore, the analysis phase can replace data flow facts by safe lower bounds if the validation of these facts is not necessary in a security scenario. The optimisation is even better applicable in the optimisation scenario because the producer can choose to drop any data flow fact if its validation becomes too costly.

The first two optimisations have already been considered in contributions which deal with special validation contexts like Java Bytecode verification. The

contribution of this chapter is to generalise the problem specific formulations and to reinterpret the ideas in the interprocedural setting.

The use of intentional under-approximation is more specific to the application scenario of this thesis because it is most useful when the consumer intends to safely apply optimisations to the program. In such a scenario, the consumer can waive any optimisation if the validation costs of the required data flow information becomes too high. In contrast, the validation in a security scenario can force the consumer to validate even those data flow facts, which require a significant amount of validation effort.

6.1 Reduction of the Certificate

A first observation which gives rise to optimisations in the validation scenario is that the validation process requires the recomputation of data flow facts for checking purposes. Some of these recomputed values can replace information in the annotations. This reduces the size of the annotations and even some checks become obsolete because the validator can rely on the self-computed values.

According to the general validation principle the validation of data flow results corresponds to the check that the given results solve the system of data flow equations which defines the data flow problem. This check requires two different kinds of tests. Firstly, the solution has to capture the local semantics of the code in a flow graph node. Thus, the output solution has to be as least as conservative as the result of the transfer function application which takes the given input solution as parameter. Secondly, the solution in the certificate has to capture the safe approximation of data flow facts at join points. Thus, an input solution has to be as least as conservative as the conservative approximation of the output solution of each predecessor node. In the intraprocedural case, the different kinds of checks correspond to the two kinds of inequalities in the following equation system

$$\begin{aligned} \forall i \in \mathit{FlowNodes}, t_i = [\![i]\!] \in T : & \\ O_i^* &\sqsubseteq t_i(I_i^*) \\ I_i^* &\sqsubseteq \begin{cases} I_{\mathit{Start}} & \text{if} \quad i = 1 \\ \bigsqcap_{j \in \mathrm{pred}_G(i)} O_j^* & \text{else} \end{cases} \end{aligned}$$

In order to check the validity of an output solution, the validator has to compute the result of the transfer function t_i with respect to I_i^* which yields a output solution $O_i^\star$. The validation of a given output solution O_i^* reduces to the check that

$$O_i^* \sqsubseteq O_i^\star = t(I_i^*)$$

However, the recomputed output solution $O_i^\star$ is already a solution which is valid with respect to I_i^*. Thus, it is not necessary to ship a solution candidate in the certificate if the validator reuses * during the validation process. This immediately reduces the size of the certificate and also avoids the check $O_i^* \sqsubseteq O_i^\star$. This idea is exploited by the KVM approach to Java Bytecode verification.

6.1.1 The KVM Approach

The "Kilo Virtual Machine" is a lightweight variant of the Java Virtual Machine which is tailored for limited devices like mobile phones. The available memory on such devices is limited to some hundred kilobytes. Therefore, the original bytecode verification algorithm - which is essentially a data flow algorithm that solves an intraprocedural type inference problem - cannot be implemented on a KVM. Thus, the Connected Limited Device Configuration [BLTY03] specifies a bytecode verification which relies on the transmission of so-called "stack maps" which contain the type information at each input of a flow node in a method.

The validation process computes the corresponding output solution $O_j^\star = t_j(I_j^*)$ and checks that

$$\forall i \in \mathrm{succ}_G(j): \qquad I_i^* \sqsubseteq O_j^\star$$

This check can be performed easily because all input solutions are available in the certificate and the offsets of the conditional branch instructions at the end of a flow node identify the successor nodes explicitly.

Nevertheless, the check differs from the check defined by the equation system of the data flow problem

$$I_i^* \sqsubseteq \bigsqcap_{j \in \mathrm{pred}_G(i)} O_j^\star$$

Essentially, the approach replaces check that an input solution is as least as conservative as the conservative approximation of all input solutions by the check that the inequality holds for each output solution separately. This strategy is justified by the observation that

$$\begin{array}{lcl} I_i^* & \sqsubseteq & \displaystyle\bigsqcap_{j \in \mathrm{pred}(i)} O_j^\star \\ & \Leftrightarrow & \\ \forall j \in \mathrm{pred}(i) & & I_i^* \sqsubseteq O_j^\star \end{array}$$

Informally, if I_i^* is as least as conservative as the safe approximation of all output solutions of all predecessor nodes, then it is more conservative than each output

solution and vice versa. Consequently, the validator can check the validity of a given input solution *successively* by checking its validity with respect to a *single* output solution.

This decomposition of the check into checks, which involve a single input and output solution only, enables the the validator to completely process a single output solution once it is computed. This is important because it reduces the intermediate storage which is required to hold the computed output solutions to a single element. At this point we can already observe that the reuse of recomputed values in the validation process has the potential to produce additional costs: the validator may need intermediate storage to keep the computed values.

The KVM approach avoids the need for additional storage by the reorganisation of the join-point checks: instead of checking an input solution against all output solutions of predecessor blocks it subsequently checks a single output solution against the input solutions of all successor blocks so that it suffices to store a single intermediate output solution only.

Interestingly, the decomposition of the join-point check requires that the validation looses the potential to validate the *maximality* of the given fix-point solution. The validation of the maximum fix-point solution requires the computation of the safe approximation of all output solutions and the check that the result is *equal* to the given input solution. The problem is that

$$\begin{array}{lcl} \forall j \in \mathrm{pred}(i) & & I_i^* \sqsubseteq O_j^\star \\ & \Rightarrow & \\ I_i^* & = & \bigsqcap_{j \in \mathrm{pred}(i)} O_j^\star \end{array}$$

Nevertheless, the loss of the potential to validate the maximality of the solution does not affect the safety of the bytecode verification. *Any* solution which passes the checks is a valid one and thus safe. A weaker result may still suffice to ensure the type-safety of the program. This is the key observation which leads to optimisations which exploit the intentional under-approximation principle (see Section 6.3).

6.1.2 The Difference Certificate Approach

The general observation that the validator can reuse data flow facts computed during the validation process can be exploited even more aggressively than in the KVM approach. The general idea is to use computed output solutions as candidates for subsequent input solutions. The certificate supplies *difference information* only if the input solution of the data flow analysis differs from the input solution candidate derived from preceding output solutions.

The idea was originally proposed by Rose in her approach to lightweight bytecode verification [RR98] and its general applicability to intraprocedural analysis results is emphasised in [Ros03].

This section reformulates the original idea in terms of the validation of data flow equations. Furthermore, we will address the additional question how long intermediate results have to be kept by modified validation process. Consider the example in Figure 6.1 which shows two different kinds of joint points of paths in the flow graph.

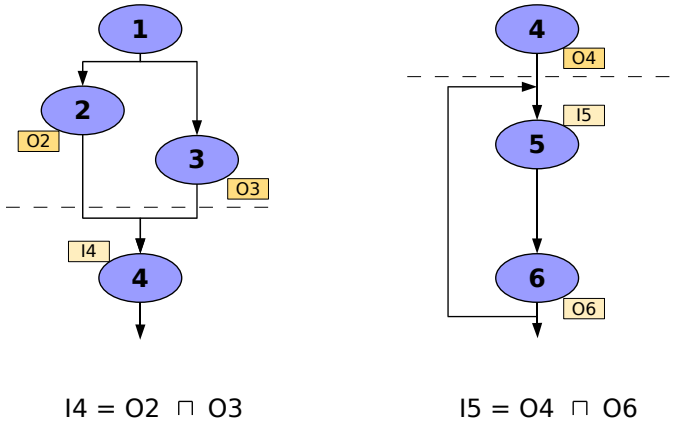


Figure 6.1: Construction of Input Solutions during the Validation Process

The situation on the left hand side shows the join of control after a conditional statement. The validator has already computed the output solutions $O_2^\star$ and $O_3^\star$. Thus, it is possible to compute the safe approximation $I_4^\star = O_2^\star \sqcap O_3^\star$. This data flow fact can immediately act as a valid value for the input solution I_4. As a consequence, the input solution does not have to be stored in the certificate and even the check becomes obsolete because the validator recomputed the input solution based on valid data flow facts.

The situation differs on the right hand side in Figure 6.1 where a backward edge contributes to the information at the input of node 5. The output solution $O_4^\star$ is already available but the output solution O_6 is not. In order to construct a suitable input solution in such a case, the validator relies on *difference information* in the certificate. This difference information serves as a substitute for the unknown terms in the equation. The validator constructs the input solution by the safe approximation of an input solution *candidate* $I_{5c}^\star = O_4$ and a difference element Δ_5^* for the flow node, thus

$$I_5^\star = I_{5c}^\star \sqcap \Delta_5^* = O_4^\star \sqcap \Delta_5^*$$

More generally, the validator can use the safe approximation of all *available* output solutions as an input solution candidate $I^{\star}_{c5}$ and the certificate contributes the effects of the safe approximation of all subsequent output solutions as a single difference element Δ^{*}_{5}.

Interestingly, the difference information is not always necessary, because if

$$I^{\star}_{c5} \sqsubseteq \Delta^{*}_{5} \Rightarrow O_4 \sqcap \Delta^{*}_{5} = O_4$$

Intuitively, if the input solution candidate is already as conservative as all information which is contributed by subsequent output solutions, then it is not modified by the difference element which in turn can be omitted from the certificate.

The problem is closely related to the fix-point computations in the iterative analysis algorithm. The data flow algorithm has to iterate only if some information which is contributed by subsequent nodes invalidates an input solution used in the preceding iteration. When the flow graph is processed in code order, this can only occur via a backward edge. Thus, difference information is only required at target nodes of backward edges and only if the input solution candidate does not already constitute the final result. Intuitively, an entry in the difference certificate predicts the result of a fix-point iteration whenever it differs from the input solution candidate constructed during the validation pass.

In the original application scenario of Java Bytecode verification back edges - which correspond to loops in the control flow graph - are rare and the type information about local variable registers[1] does not change very much during the analysis process. Thus, the conservative approximation of the output solution of predecessor nodes visited during the traversal in code order usually corresponds to the final solution already - and the difference certificate becomes empty.

However, the reduction to a difference certificate imposes a new challenge, because the validator has to keep intermediate results in some temporary storage. The required size of such temporary storage is crucial for a target device like a smart card because random access memory is usually a much more valuable resource than EEPROM where the certificate can be stored.

Recall that the validator has to check that

$$\forall j \in \mathrm{pred}_G(i): \qquad I_i \sqsubseteq O_j$$

Thus, the situation on the right hand side in Figure 6.1 requires that $I^{\star}_{5} \sqsubseteq O^{\star}_{4}$ and that $I^{\star}_{5} \sqsubseteq O^{\star}_{6}$ hold. The first check immediately holds, because $O^{\star}_{4}$ was used to construct $I^{\star}_{5}$. However, the second check is still pending, because the validator must not assume that the difference information in the certificate contains the

[1] In contrast to the Virtual Machine Specification we denote the local variables in the method frame of a virtual machine as local variable registers to separate them from the local variables of the source language.

correct value. Therefore, the input solution $I_5^\star$ has to be kept in memory until the output solution $O_6^\star$ becomes available.

The original approach of Rose stores the input solution for all flow nodes which are the target of a backward edge. Additionally, output solutions are kept as long as they contribute to the computation of input solution candidates - i.e. as long as the last successor node has been processed. This leads to a memory consumption which depends on the overall number of backward edges and to the maximum number of forward edges which pass a cut in the flow graph.

However, the observation that output solutions have a limited lifetime extends smoothly to the input solutions which are stored for subsequent checks: An input solution can be released as soon as the last check has been performed - i.e. when the last predecessor node has been processed. The following section establishes a model for the lifetime of data flow values and deals with the minimisation of the number of intermediate results.

6.2 Lifetime of Data Flow Facts in the Validation Process

In Section 6.1 we observe that the size of the transmitted certificate can be reduced significantly if the validator reuses data flow facts which are computed during the validation process. Essentially, difference elements have to be transmitted only if the final result of the iterative fix-point computations differs from the solution candidate constructed by the validator. However, the strategy requires that the validator stores some of the data flow results as long as their validity has finally been established.

In this section we will develop a graph model for the dependencies between data flow facts and show how it is possible to estimate the maximum number of intermediate results during the validation process by an inspection of the graph. Next, we reinterpret the different optimisations in terms of the dependency model. Furthermore, the graph model reveals that the number of intermediate results depends on the order in which the validator processes the flow graph nodes. This offers an additional optimisation opportunity as discussed at the end of the section.

6.2.1 Dependency Model

The question how long a data flow fact is needed during the validation process is closely related to the dependency between data flow facts. These dependencies can be easily derived from the system of data flow equations. Consider the flow graph in Figure 6.2.

The corresponding system of data flow equations shows that there is exactly one defining equation for each data flow fact. Furthermore, several data flow facts can contribute to the computation of a specific value like the output solutions

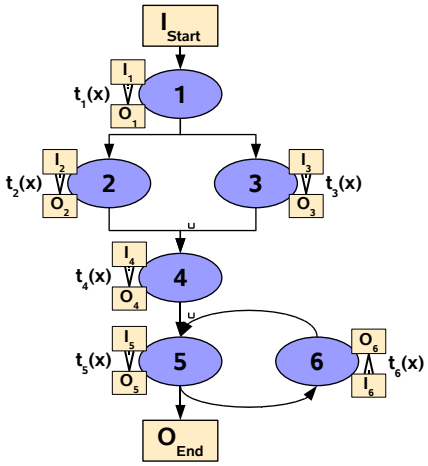


Figure 6.2: Intraprocedural Flow Graph

O_2 and O_3 give rise to the definition of I_4. Likewise a single data flow fact can contribute to the definition of several data flow facts like O_1 which contributes to the definition of I_2 as well as I_3.

$$
\begin{aligned}
I_1 &\sqsubseteq I_{Start} \\
O_1 &\sqsubseteq t_1(I_1) \\
I_2 &\sqsubseteq O_1 \\
O_2 &\sqsubseteq t_2(I_2) \\
I_3 &\sqsubseteq O_1 \\
O_3 &\sqsubseteq t_3(I_3) \\
I_4 &\sqsubseteq O_2 \sqcap O_3 \\
O_4 &\sqsubseteq t_4(I_4) \\
I_5 &\sqsubseteq O_4 \sqcap O_6 \\
O_5 &\sqsubseteq t_5(I_5) \\
I_6 &\sqsubseteq O_5 \\
O_6 &\sqsubseteq t_6(I_6) \\
O_{End} &\sqsubseteq O_5
\end{aligned}
$$

We can capture these dependencies in a graph which contains the data flow facts as nodes. A directed edge (n_1, n_2) connects two data flow facts n_1, n_2 if n_1 contributes to the defining equation of n_2. The graph in Figure 6.3 captures the direct dependencies in the data flow equation system.

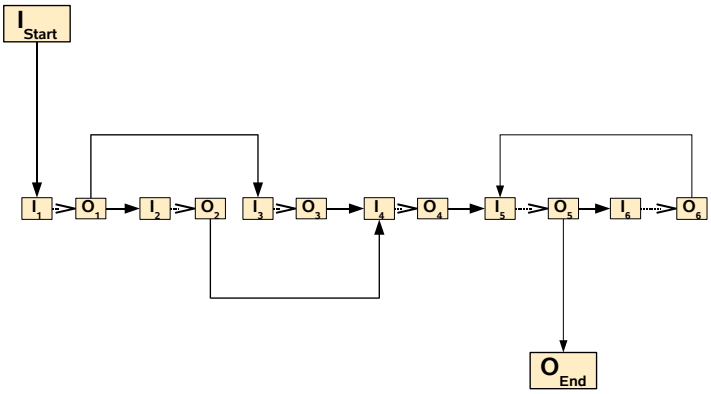


Figure 6.3: Dependence Graph

This graph is closely related to the original flow graph because the flow edges directly correspond to edges which connect output to input solutions. The other kind of edges which connect input to output solutions directly corresponds to the nodes of the flow graph. The advantage of the new representation is that it abstracts from the different kinds of solutions so that we can argue about the lifetime of input and output solutions in a uniform way.

The validation process checks that a given solution solves the system of data flow equations. The validator can perform these checks easily if all data flow facts are available. However, this simple strategy requires that the whole solution is kept in memory during the validation process. Thus, it is an important question, how long a data flow fact is still required and when it can be dropped. The answer is given by the dependence graph.

The validation process is a linear pass over the system of equations and the dependence graph respectively. A single data flow fact can be checked as soon as all predecessor nodes in the dependence graph have been visited. Similarly, the value itself is needed as long as there exists an unprocessed successor node. Thus, the lifetime of a data flow fact starts when it is processed and ends after the processing of the last successor or predecessor respectively.

Consequently, the lifetime of a data flow fact depends on the order in which the solutions are processed and captured within a linear ordering of the dependence graph.

6.2.2 Reuse and Check

We will now reconsider the different validation strategies in terms of the dependence graph model. To begin with, we can observe that both the KVM and

the difference certificate approach check the dependency between two solutions separately. For example, the check of the defining equation of $I_4 \sqsubseteq O_2 \sqcap O_3$ is not performed by first applying the conservative approximation of O_2 and O_3 and a subsequent check. In contrast, the check is split into two checks, which finally establishes the validity of the equation because

$$I_4 \sqsubseteq O_2 \wedge I_4 \sqsubseteq O_3 \Rightarrow I_4 \sqsubseteq O_2 \sqcap O_3$$

The edges in the dependence graph directly correspond to these checks which in turn justifies that a data flow fact can be released as soon as all checks involving this element have been performed. The fact that the validator can take the lifetime of data flow facts into account reduces the number of data flow facts which have to be kept in memory during the validation process.

Additionally, the difference certificate approach uses a second principle in order to reduce the amount of information which has to be transmitted in the certificate: Instead of checking a direct dependency between two data flow facts, the validator can use an available data flow fact as solution candidate for a dependent solution. This candidate fulfils the properties of the check by construction.

Consider the situation for the input solution of node 5 in the example:

$$\begin{aligned} O_4 &\sqsubseteq \ldots \\ I_5 &\sqsubseteq O_4 \sqcap O_6 \\ \ldots & \\ O_6 &\sqsubseteq \ldots \end{aligned}$$

The output solution O_4 is available already. Thus, it can act as a solution candidate for I_5 and the check that the value of I_5 is smaller or equal than O_4 holds trivially. Furthermore, the certificate has to contribute difference information if and only if the value of O_6 weakens the result of the conservative approximation expression.

In fact, the strategy of the difference certificate approach can be reformulated as follows: A solution candidate for a data flow fact is constructed from the defining data flow by substituting variables by available values and by values provided in the certificate.

A forward edge in the dependency model indicates that the solution of the source node is available and can be integrated into a solution candidate for the target node. The solution candidate is used when the validator processes the target node. Thus, the length of a forward edge determines the lifetime of a solution candidate.

Similarly, backward edges represent postponed checks. The validator has to check the solution of the source node against the solution of the target node. Therefore, the solution which has been derived for the target node has to be kept in memory until the solution of the source node becomes available.

6.2.3 Optimisation Goals

The key observation of the preceding sections is that the validator can process the data flow equations in an arbitrary order. However, the processing order defines forward and backward edges. Thus, the number of reusable results, deferred checks, and the lifetime of data flow facts changes according to the processing order. This observation can be used to improve the validation process further. However, the consumer has to be capable to deal with an arbitrary validation order which is determined at the producer site. This general idea has already been observed in [KK05] but it has not been generalised to the interprocedural setting.

The validation process can be optimised in two different ways. Firstly, the reuse of data flow facts computed during the validation process reduces the size of the transmitted certificate. Secondly, the processing order determines the maximum number of intermediate solutions which have to be kept in memory during the validation process.

A straight-forward idea to achieve the first optimisation goal is to minimise the number of backward edges in the linear arrangement of the dependence graph. A backward edges models the fact that the current data flow fact depends on a fact which has not been computed yet. Thus, the certificate has to supply difference information if the unknown data flow fact weakens the solution candidate which can be derived by the evaluation of the defining equation with the available data flow facts. This shows, that the number of backward edges is only an indirect criteria for the size reduction of the certificate, because some backward edges may not trigger the inclusion of difference information.

A depth-first traversal is a good choice to minimise the number of backward edges. Especially, if the flow graph is reducible then the set of backward edges is independent from the chosen depth-first traversal [HU74]. Intraprocedural flow graphs are usually reducible so that the strategy is very reasonable in the intraprocedural setting. However, the situation changes for irreducible graphs because the number of backward edges in irreducible graphs does depend on the order of the traversal [CHK04]. Nevertheless, the depth-first strategy still provides a good starting point for the certificate reduction. It is important to observe that this optimisation is performed at the producer site. Thus, complex optimisations strategies can be acceptable because the primary goal is to relieve the consumer site from computation costs.

The second optimisation goal is the reduction of the maximum number of intermediate results in the validation process. A first idea is to optimise the maximal cut in the linear arrangement [KK05] because it is an upper bound of the required number of intermediate results. A cut in the linear arrangement separates processed data flow results from unprocessed ones. Forward edges which cross the cut indicate that an available data flow fact contributes to the computation of a unknown data flow fact. Thus, the available fact should be stored to provide a solution candidate for the unknown fact. Similarly, a backward edge which crosses the cut indicates that an available solution

depends on an unknown fact. Thus, the target fact has to be kept in memory until the corresponding check has been performed.

The real costs differ slightly from this cost measure because some intermediate results are counted several times. Consider the situation in Figure 6.4.

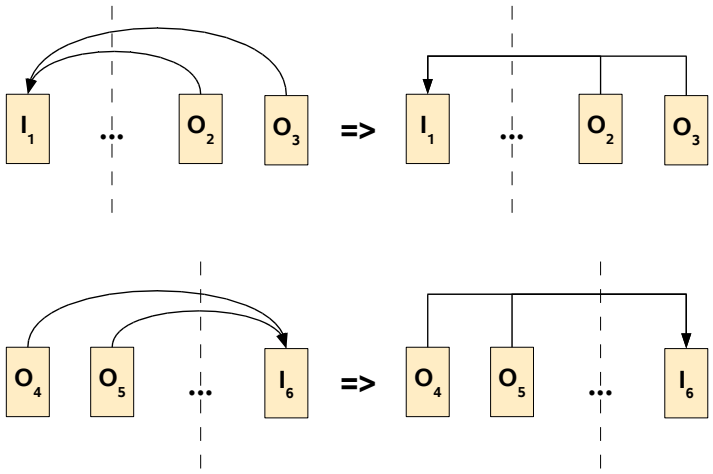


Figure 6.4: Multi Edges in the Dependence Graph

The input fact I_1 is the target of two different backward edges originating in O_2 and O_3 respectively. Thus, the input fact I_1 has to be stored until its last predecessor O_3 has been processed. Nevertheless, only a single place in memory is required to store the value of I_1 even though there are two backward edges cross the cut between I_1 and O_2.

The same situation arises when a solution is the target of two incoming edges like I_6. An input solution candidate for $I'_6 = O_4 \sqcap O_5$ can be computed as soon as the output fact O_5 is available. Again a single place in memory suffices to store this candidate even though two forward edges cross the cut between O_5 and I_6.

The optimisation algorithm can take this observation into account if it combines forward and backward edges with the same target into a special multi-edge as depicted in Figure 6.4. Such a multi-edge is counted only once when the number of cut-crossing edges is determined. This technique is also used by approaches to the register allocation problem in classical compilers [Bel66], [BCT94], [Cha82].

This model improves the significance of the cost measure but it complicates the search for a linear arrangement that provides the minimal cut. The most precise cost model is even more complex if the validation process takes advantage of an additional degree of freedom. The example in Figure 6.4 assumes that output solutions are immediately used to construct an input solution candidate. This strategy reduces the storage requirements if the candidate depends on several

available output solutions. However, Figure 6.5 shows, that this approach does not always yield the best result.

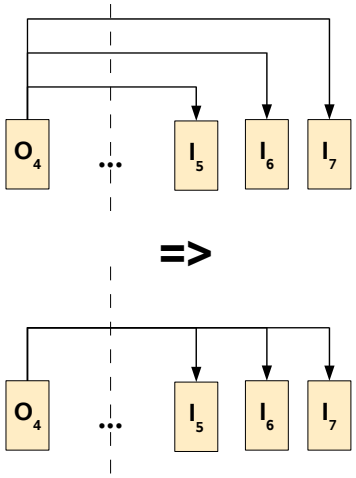


Figure 6.5: Storage Requirements in Presence of Multiple Successors

The immediate construction of the input solution candidate for $I_5 \dots I_6$ leads to the storage of three candidates. However, it suffices to construct the input candidates when they are needed. Thus, the validator can keep the output fact O_4 in a single storage location. Figure 6.5 shows the different kinds of edges which model the two strategies.

It is important to exploit this optimisation opportunity in the intraprocedural scenario because switch-statements produce a significant amount of successor nodes in the control-flow graph. The immediate construction of input solution candidates produces one copy for each branch in the switch-statement while the reuse of the output fact gets along with a single element.

All in all, the optimisation of the intermediate storage requires both, the choice of a reasonable linear arrangement and a flexible construction of input candidates. However, the whole optimisation scenario is even more complex, because the choice of the linear arrangement also influences the size of the certificate. Therefore, a reasonable heuristic is to choose a depth-first traversal which minimises the number of backward-edges and to optimise the memory allocation during this traversal using the ideas described in this section. Nevertheless, more complex optimisation strategies can be applied to the problem, because the effort is spend solely on the producer site.

6.3 Safe Lower Bounds

Section 6.1 and 6.2 describe optimisation strategies for the size of the certificate and the maximal number of intermediate results respectively. Both approaches reduce the cost for the validation of data flow solutions.

The memory requirements of the validation process can be reduced even further, if we exploit knowledge about safe-lower bounds in the analysis result. The most pessimistic element of the data flow lattice represents the loss of all information about the corresponding data flow fact. Thus, it does not have to be stored explicitly but can be implicitly assumed whenever data flow information is omitted. Special data structures can use this idea to reduce memory consumption as discussed in Section 8.2.4.

The most pessimistic element $\bot$ is always a safe-lower bound. However, some analysis supply more precise safe-lower bounds. For example, a type inference analysis for Java programs (see Section 7.3) can use the *declared types* of fields and of result types of method invocations to recreate omitted type information. The general principle is the same: only valuable information is stored explicitly in the annotation or in intermediate results, while omitted values are reconstructed from safe-lower bounds if needed. The key insight is that the validator can immediately trust a safe-lower bound.

Obviously, the potential of this optimisation directly depends on the quality of the analysis results. More precise results require that more data flow facts have to be stored explicitly while the storage requirements decrease the more data flow facts have been reduced to a safe-lower bound during the analysis. Interestingly, the optimisation is *adaptive* in the sense that it offers a trade-off between the quality of the solution and the memory needed to store the solution. This can be exploited in several ways.

6.3.1 Lattice Strength Reduction

We model the program state in terms of a data flow environment which is a mapping from data flow variables to data flow values. Each data flow variable represents a single piece of information a specific analysis is interested in. The size of such an environment depends on the number of data flow variables the analysis has to track.

For example, a constant propagation analysis tries to determine whether a local variable contains a constant value or not. Thus, the maximum number of local variables in a method limits the size of the corresponding data flow tuple. The size increases if the analysis additionally takes global variables into account. Conceptually, a data flow environment can be extended point-wise in the same way a power-set lattice is extended by the inclusion of an additional element.

A straight-forward implementation of the analysis tracks *all* variables because any of them has the potential to store a constant value. However, only a very small number of variables will contain a constant value. Thus, the analysis could

have been reduced to those variables which actually contain constant values if the result of the analysis would have been known beforehand. An analysis cannot benefit directly from this observation, but the validation process can. The information, which variables actually have to be considered to establish the validity of the result can be shipped along with the annotations. The validator can use the information to streamline the data-structures which are used to store the data flow environments. Conceptually, the validator can reduce the power-set lattice of all variables to the power-set lattice of relevant variables. This is why we call this technique "lattice strength reduction".

Bernardeschi [BLMM05] suggests a similar idea to reduce the memory requirements of Java Bytecode verification. Essentially, the type check is split into several phases which deals with different kinds of types like integer and reference types separately. As a consequence, each phase has to deal with a reduced number of facts only which reduces the maximum memory consumption.

The savings by lattice strength reduction can be significant for analysis like constant propagation which usually leads to a small amount of valuable information. Furthermore, the technique applies well to analyses which exhibit strong lower bounds like the declared types of fields and methods in Java programs. However, the efficiency of the approach is limited if the analysis derives potentially useful information for each point in the program. For example, the computation of available expressions will determine that each expression is at least available directly after its computation.

Nevertheless, the technique is still valuable to reduce the size of the lattice *intentionally* as discussed in the following section.

6.3.2 Intentional Under-Approximation and Demand-Driven Analysis

So far, we have observed that safe lower bounds and the reduction of the data flow lattice to relevant elements improve the efficiency of the validation process. This observation becomes even more important if we take into account that the analysis results shall serve a specific purpose in our application scenario. The analysis always tries to compute the strongest result possible. In contrast, the validator only has to check the weakest result required to prove that the program respects a security policy or that an optimisation can be applied safely.

Thus, the producer can weaken a strong analysis result before it is transmitted to the consumer. The weaker the analysis result is the more efficient the safe lower bound or lattice strength reduction techniques become. For example, the results of an available expression analysis can be reduced to those expressions which actually contribute to an expression which offers an *optimisation opportunity*. All other expressions can be omitted by a reduction of the corresponding data flow lattice to the relevant expressions.

This general technique can be applied to the security and the optimisation scenario. However, it is more effective, if the analyses results are used to apply

optimisations only. The reason is that the producer can even weaken the results if this implies that some optimisation opportunities are lost because missed optimisations do not break the integrity of the consumer. In contrast, the result can only be weakened up to specific bound in the security scenario. If the consumer uses the analysis result to enforce a specific property of the program, then the results have to be strong enough to enable the corresponding checks.

The question whether or not a data flow fact is relevant is a transitive property. Not only the data flow facts which are immediately required to justify an optimisation or to prove a security policy but also all data flow facts the fact depends upon. Therefore, the result cannot be weakened arbitrarily.

The problem to decide which analysis results are relevant is closely related to demand-driven analysis. This kind of data flow analysis techniques starts from a given program point and analyses only those pieces of the program which correspond to the data flow facts at this point. The producer can use a demand-driven analysis to determine the weakest results that still guarantee the properties demanded by the validator.

6.4 Reinterpretation in the Interprocedural Scenario

The main concepts for the optimisation of the validation process can be summarised as follows:

- The system of data flow equations contains a single defining expression for each data flow fact.
- The immediate dependencies between a data flow fact and the facts which contribute to its defining equation can be modelled in a dependence graph.
- The validation pass corresponds to a linear arrangement of the dependence graph.
- The certificate has to contain difference information only if a final data flow fact is weaker than the currently available facts suggest. This can only apply if a data flow fact is a target of a backward edge in the linear arrangement of the dependence graph.
- The linear arrangement of the dependence graph also determines the lifetime of a data flow fact. A fact can be dropped when the last immediate successor and predecessor has been processed.
- The maximal cut in the linear arrangement is an upper bound for the maximal number of intermediate results during the validation process. This number can be reduced by a flexible construction of input solution candidates which corresponds to the question when an available data flow fact is substituted into the defining equations it contributes to.

These general observations can be reinterpreted in the interprocedural setting.

6.4.1 Dependencies in the Interprocedural Result

The validation of interprocedural summary functions can be modelled by a linear arrangement of a dependency graph like the validation of intraprocedural results. The validation pass visits each function and within each function all flow graph nodes. Therefore, the linear arrangement contains a node for each intraprocedural summary function of each method in the program.

The intraprocedural control flow edges connect summary functions to each other like they connect data flow values to each other in the intraprocedural case. However, the interprocedural dependence graph contains additional edges due to the calling relations between methods.

Intraprocedural analyses use a constant transfer function t_{call_i} to safely approximate the potential effects of a call. In contrast, the computation of summary functions integrates the summary function ψ_n of the callee n:

$$O_i^* \sqsubseteq t_{call_i}(I_i^*) \qquad \text{intraprocedural case}$$

$$\psi_{i'}^* = \psi_n^* \circ \psi_i \qquad \text{interprocedural case}$$

Thus, the output summary $\psi_{i'}$ does not only depend on the input summary but also on the interprocedural summary of the callee. Therefore, the complexity of the dependence structure increases as depicted in Figure 6.6.

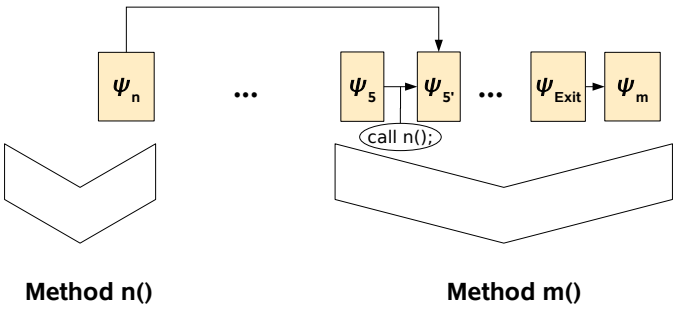


Figure 6.6: Interprocedural Dependencies

In the example, the output summary $\psi_{5'}$ does not only depend on the intraprocedural input summary ψ_5 but also on the summary function of the callee ψ_n.

Such an interprocedural summary function of a whole method corresponds to the output summary function of the exit node of the method and the dependency model has to be adopted appropriately.

Similarly to the intraprocedural case the linear arrangement of the summary function nodes constitutes a potential validation order and the dependence edges describe which data flow facts are already available. For example, the forward edge from the exit node of method n states that the interprocedural summary function is required for the validation of the intraprocedural summary function $\psi_{5'}$. We will now reinterpret the different optimisation strategies on the interprocedural dependence graph.

6.4.2 Difference Certificates

The key idea of difference certificates is to reuse information which is computed during the validation phase. Available information corresponds to forward edges in the dependence graph. Therefore, it is a reasonable strategy to apply a depth first traversal within each method, like in the intraprocedural case. The dependency edges which model the intraprocedural control flow introduce backward edges for loops only. The safe approximation of the summary functions of known predecessors can act as a solution candidate for the input summary function of the loop header. Thus,

$$\psi_{i'} = \bigsqcap_{j \in \mathrm{pred}(i) \wedge j < i} \psi_{j'} \sqcap \psi_{\Delta_i}$$

$$\text{where} \Delta_i = \bigsqcap_{k \in \mathrm{pred}(i) \wedge k > i} \psi_{k'}$$

Once again, the difference information is omitted if it is already safely approximated by the known output summary functions.

Interprocedural dependence edges behave differently. The validation pass processes all methods in some order. It is important to note, that the interprocedural dependence edges flow from the *callee* to the call site in the *caller*. Thus, a linear arrangement which processes a callee before all of its callers produces forward edges, so that no difference information is needed. This is not surprising because a validation pass which starts at the leaf nodes and proceeds in a bottom-up order through the call graph has all required summary functions at hand.

Nevertheless, cycles in the call graph introduce additional backward edges in the linear arrangement of the summary functions which differ from the intraprocedural backward edges. An interprocedural backward edge indicates that the validator processes the equation of a call node when the summary function of the callee has not yet been constructed. Thus, the validator tries to check that

$$\psi_{i'} \sqsubseteq \psi_{call_i} \circ \psi_i$$

without a solution candidate for ψ_{call_i} at hand. This situation differs from the situation at intraprocedural join points because composition of an arbitrary summary function can lead to any result. Therefore, it is not possible to integrate some kind of difference information in the certificate. In contrast, the full summary of the callee has to be shipped in the certificate if a caller is processed before the callee. However, the summary function of a specific callee has to be shipped only once and can be reused at subsequent call sites.

The final question with respect to the difference approach is how the producer can construct efficient Δ-functions for intraprocedural join points. Fortunately, difference functions can be derived from the summary function model easily as discussed in Section 8.3.3.

6.4.3 Intermediate Results

The general observations about the lifetime of data flow facts during the validation process apply directly to the validation of summary functions as well. Each output summary function is relevant until the last successor node in the intraprocedural flow graph node is processed. Similarly, input summary functions have to be stored for subsequent checks, whenever their flow graph node is the target of a backward edge in the linear arrangement. Such input solutions are relevant until the last predecessor node is processed by the validation pass.

Additionally, the number of intermediate summary functions can be reduced further by the same flexible substitution techniques, which have been considered for the intraprocedural validation scenario.

In contrast, flexible substitution strategies cannot be applied to interprocedural summary functions and they do not give rise to solution candidates for successor nodes in the dependence graph either. The problem is that the interprocedural summary function is used differently at a call site. It acts as transformer of an unknown input summary function ψ_i, because

$$\psi_{i'} \sqsubseteq \psi^{\star}_{call_i} \circ \psi_i$$

This equation cannot be exploited to construct a solution candidate if one of the participating summary functions is missing. However, it is still possible to compute the output summary if $\psi^{\star}_{call_i}$ and ψ_i have already been constructed during the validation pass.

Furthermore, the dependency graph determines the additional lifetime constraints in the interprocedural validation scenario. An interprocedural summary has to be kept in memory until the last call site has been processed. Additionally, the intraprocedural output summary functions have to be stored

for a subsequent check if the summary function whenever a caller is processed before the callee.

Thus, the general observation that it is advantageous for the validator to process the analysis result in an order which minimises the number of backward edges applies for interprocedural dependency edges as well. The interprocedural dependency edges encode the call graph of the program in reverse order because the edges originate in callees and target the callers. Thus, the order which avoids backward edges roughly corresponds to a bottom-up traversal of the call graph that starts at leaf methods and subsequently proceeds to the callers. Backward edges only arise due to recursive parts of the call graph. As the structure of call graphs is usually not reducible, the order in which the different methods are processed influences the number of backward edges. However, the consumer has enough computational capabilities to find a good solution for this optimisation problem.

All in all, the storage of intraprocedural summary functions can be managed by the same strategies which have been invented in the intraprocedural setting. Additionally, a single copy of each interprocedural summary has to be kept in memory until the validator has past the last call site of the corresponding method. Thus, the flow nodes of each method should be processed according to the strategies of the intraprocedural scenario and the methods themselves should be arranged according to a bottom-up traversal of the call graph to minimise the lifetime of callee summaries.

6.4.4 Modular Results and the Dependence Graph

The central idea for the construction of the dependence graph, is that the defining equation of each data flow fact connects the fact to the data flow values it depends upon. We can apply the same idea to modular results.

Consider the example in Figure 6.7 and assume that the analysis in question performs a simple copy constant propagation. The modular representation of the summary function of this method is constructed by function composition and the safe approximation of the summary functions which capture the two control flow paths through the method. The summary results to

$$\psi_m = \langle \ldots, e^a, \ldots \rangle = \langle \ldots, \bot \ldots \rangle$$

We can consider this equation to be the definition of the modular result for ψ_m. Consequently, the external variables show on which other data flow facts the summary depends upon. The modular summary function ψ_m does not contain a reference to the callee n anymore, although the method m *does* call method n.

This reason is that the normalisation process modifies the equation system. The effect originates from the safe approximation of the summary functions $\psi_{2'}$ and $\psi_{3'}$ at the join point. On the left branch the value of a is known not to be

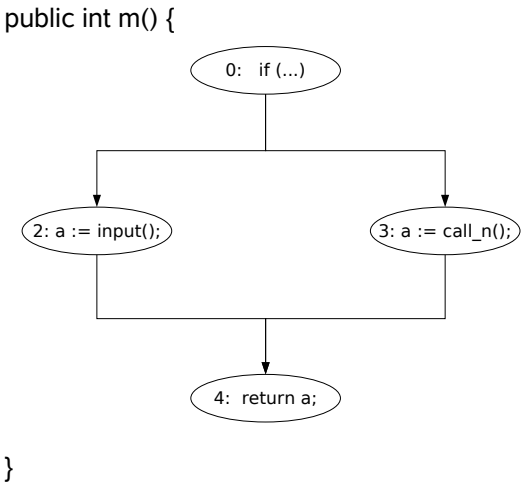


Figure 6.7: Summary Functions of Copy Constant Propagation

constant while its value depends on the invocation of method n on the right branch. Thus,

$$\begin{aligned} \psi_4 &= \psi_{2'} \sqcap \psi_{3'} \\ &= \langle \ldots, \bot \sqcap s_n(\ldots), \ldots \rangle \\ &\xrightarrow{BSC} \langle \ldots, \bot, \ldots \rangle \end{aligned}$$

Essentially, the $\xrightarrow{BSC}$-reduction removes the dependency because a safe lower bound on one path subsumes the potential influence of the invocation of a callee on the other path.

As a consequence, the validation pass does not have to keep the summary function of method m in the intermediate storage until the callee n has been processed. Essentially, the partial evaluation strategy that is encoded in the normalisation has suppressed the analysis of specific program paths if other program paths already lead to some safe lower bound. This technique reduces the dependencies between data flow facts so that some intermediate results can be dropped ahead of time.

Obviously, the benefit of the technique depends on the question how many pieces of the result consist of or are intentionally weakened to a safe lower bound. Nevertheless, the dependency model described in this section is capable to deal with the specific properties of modular results smoothly.

6.5 Summary and Related Work

In this chapter we considered the dependencies between data flow values and several optimisation strategies for the validation process. The defining equations of the data flow values define a dependence graph. The data flow elements form the nodes of this graph and two data flow elements are connected by an edge if the fact of the source node is used in the defining equation of the target node.

The validation pass is a linear arrangement of the dependence graph. Forward edges model that data flow facts which define another fact have already been processed. In contrast, backward edges indicate that a data flow fact which contributes to the definition of the element under consideration has not been visited yet. This model is suitable to explain different optimisation strategies of the validation process.

Firstly, the idea to store only difference information in the certificate directly relates to the number of backward edges in the linear arrangement of the dependence graph. Difference information has to be supplied only if there is a backward edge and if the data flow information contributed by the edge is not already subsumed by available contributors.

Secondly, data flow elements have to be kept in memory only as long as the last predecessor and the last successor node have been processed. The maximal number of cut edges in the linear arrangement of the dependence graph is an upper bound for maximal number of intermediate elements required during the validation process. The validator is free to either keep intermediate results in storage or to merge them into the defining equations they contribute to. It is possible to reduce the intermediate elements further, if the validator makes use of this possibility.

The third way to reduce the costs of the validation process is to validate a weaker fix point than the maximal one. The intention is that weaker data flow results can be represented more efficiently and that weaker results exhibit a simpler dependency structure.

The different optimisation strategies directly apply to the interprocedural dependence graph as well but the summary functions of callees impose additional challenges. Firstly, the summary functions have to be supplied completely, because they act as function transformers and do not contribute to a conservative approximation like intraprocedural functions at join points in the control flow. Secondly, the callee summaries have to be kept in memory until the last call site is processed. In the incremental validation scenario it is possible to reduce the number of intermediate *open* summary functions by an adoption of the dependence graph model. Essentially, the normalisation process has the potential to remove dependencies on other data flow facts, so that open summary functions can be dropped earlier than usual. However, this strategy applies mostly to analyses which exhibit a larger number of safe lower bounds in the final result.

The idea to store difference information in the certificate stems from the lightweight approach to Java Bytecode verification [RR98] and has been adopted

in the abstraction carrying code approach as well [AASPH06]. The fact that the difference information depends on the traversal strategy during the validation process has been observed in [AASPH06], but the question which traversal strategies to choose is not considered. Amme et al. observe in [Amm07] the correlation between backward edges and the points where the certificate might have to contribute additional information. They suggest a reversed postorder traversal, which minimises the number of back edges for reducible intraprocedural flow graphs, but which may not be the best choice of irreducible graphs. Furthermore, the approach does not exploit the fact that the validation pass has already computed a solution candidate so that not all potential annotation points have to contribute information.

To our knowledge, the reduction of the maximum number of intermediate results during the *validation process* has only been addressed in [KK05]. The dependence graph model presented in this chapter generalises the approach to the interprocedural setting. Bernardeschi suggests an approach to reduce the number of intermediate results during the intraprocedural analysis phase [BFM06]. Essentially, the postdominator relation of nodes in the control flow graph is used to decide when intermediate results can be safely dropped because they are not needed to analyse the subsequent flow graph nodes.

An incremental approach to validation of data flow results is discussed in [AAP06]. Essentially, the capability of the underlying constraint solver system to deal with incremental extensions of a problem definition is exploited to adopt the abstraction carrying code approach. However, the extension impacts the effectiveness of the difference certificate approach because larger descriptions of data flow equations have to be shipped. Furthermore, the organisation of the validation process is delegated to the constraint solving system so that it is not clear if special knowledge about the structure of the data flow problem is exploited.

Recently [RSX08], Rountev presented a combination of his approach to the analysis of large software libraries [Rou05], [RKM06] and the framework for the analysis of IDE problems by Reps [SRH96]. The central idea is that summary functions which do not contain references to external methods are subsequently inserted into the summary functions of their callers. This corresponds to a bottom up traversal of the call graph. However, the approach does not try to estimate the influence of callees on the summaries of the callers to reduce external dependencies further. Instead, it keeps a set of summary functions which explicitly define the mapping from the the start node and the return nodes of unknown callees to the exit nodes and call nodes. In contrast, the summary function model presented in Chapter 5 integrates references to external callees explicitly as function variables into the function model. This way, the function representation can be reduced to those effects of external callees which influence the result function of the caller.

7 Validatable Program Analyses

This chapter describes how the generic summary function model targets several well known analyses. The complexity of the analyses considered ranges from simple bit-vector analyses like reaching definitions to type inference algorithms in the presence of dynamic method dispatch and partially known class hierarchies.

The goal of the discussion is twofold. Firstly, we want to show how to use the generic summary function model for the specification of several data flow analyses. The specification always consists of two different parts: the specification of the inducing lattice which represents data flow values and their safe approximation and the specification of instruction-level summary functions. The framework supplies default implementations for all other pieces of an interprocedural data flow problem. Secondly, the discussion shows how the characteristics of the various data flow analyses influence the complexity of the summary function representation. Essentially, we show why the specification of simple bit-vector problems leads to simple summary functions and why more complex analyses lead to more complex summary functions.

The key observations can be roughly summarised as follows:

Separable Bit-Vector Analyses Classical bit-vector analyses use only a restricted subset of the elements of the generic summary function model. Essentially, the inducing value lattice consists of the extremal elements $\top$ and $\bot$ only. Furthermore, separable data flow analyses do not introduce any dependency between different elements of the data flow environment. As a consequence, the normalisation rules reduce a defining expression of a variable x to either $\top$, $\bot$, or x. Therefore, the representation of summary functions stays linear in the size of the environment.

Non-Separable Bit-Vector Analyses Data flow facts depend on each other in a non-separable analysis. Such dependencies on several data flow facts are captured by several data flow variables in a defining expression. However, the dependencies can usually be expressed in terms of lattice operations only, so that problem specific function application expressions are not needed. Therefore, the function representation can become at most quadratic in the size of the environment. However, this is highly unlikely, because the impact of the normalisation and the fact that the dependencies between data flow variables are sparse usually, leads to a linear size of the summary functions again.

Complex Lattices The constant propagation lattice is a prominent example for a lattice which contains more elements than the two extremal elements in

the "boolean" lattice which is the elementary building block of bit-vector lattices. The data flow expression model treats the generation of all lattice elements by constant expressions. Such expressions are subject to constant folding. Therefore, only a single constant remains in each expression. This is why the summary functions for copy constant propagation stay as simple as the summary functions for non-separable bit-vector analysis. Even the positive impact of the most pessimistic element on the size of the function representation is still significant because many variables values are not constant.

Function Application Expressions Linear constant propagation is the first problem which requires function application expressions for the specification of instruction-level summary functions, because the semantics of arithmetic expressions cannot be expressed directly in terms of the safe approximation of data flow variables. The potential impact of elementary functions is massive, because they increase the upper-bound for the representation of summary functions to $O(n^{2^d})$ where d is the nesting depth of the expressions. However, either problem specific properties or the effect of some of the normalisation techniques reduce the size of the representation again to the at most quadratic but usually linear case.

Object-Oriented Features Finally, we consider how the framework can be instantiated to specify a type inference analysis which is vital to deal with dynamic method binding in object-oriented programs.

All interprocedural analyses require the determination of all potential targets of a method invocation at a specific call site. A so-called call graph is a data structure which expresses this information. Function pointers or the closely related dynamic method dispatch of object-oriented programs introduce a cyclic dependency between call graph construction and interprocedural data flow analysis: the call-graph is required to perform any interprocedural analysis and an interprocedural type inference analysis is required to restrict the dynamic type of a call site as much as possible. The usual approach to deal with this issue is to interleave the type inference analysis and the call graph construction until a common fix-point is reached.

Interestingly, the validation of such a type inference result can be performed very easily because the validator is not aware of the interleaved fix point computations but merely checks the validity of the type inference result with respect to the implied call graph.

The validation of this sophisticated analysis is not only important because it is a prerequisite for any interprocedural analysis but also because it exhibits some additional challenges for the incremental and partial validation scenario. If the program is not completely available the approach has to cope with expandable class hierarchies and incomplete supertype relations. This is modelled within the inducing data flow lattice so that the summary function model again contains a single reference to a data flow value in each expression.

Additionally, the specification requires two elementary transfer functions to capture the semantics of array accesses and explicit type casts. However, it is highly unlikely that these elementary transfer functions ever lead to nested expressions. Thus, the summary function model stays efficient even for this quite complex analysis.

All in all, one of the key challenges for the specification of an analysis is to keep the size of function application expressions under control. We can achieve this goal in different ways. First of all, the semantics of instruction-level summary functions should express as much of the problem's semantics in terms of the core model. Secondly, the normalisation rules reduce the occurrence of function application expressions automatically, if the analysis yields safe lower bounds often. This is for example the case in a constant propagation analysis. Additionally, problem-specific properties of the elementary summary functions can either reduce nested function applications to smaller ones or they can render the occurrence of nested expressions unlikely.

However, the nesting depth is always an issue for open summary functions because function variable expressions cannot be evaluated so that the potential influence of the normalisation techniques decreases. This is a challenge mainly for the analysis phase because the final result for an analysis will not contain any function variables anymore. The open representations for intraprocedural summary functions which are additionally shipped to the consumer will not contain highly nested function variable expressions, if the size of methods is small and if loops do not immediately create a cyclic self-dependency for a call site.

7.1 Bit-Vector Analyses and the Power-Set Lattice

A bit-vector analyses is a simple kind of a data flow analyses because it just computes whether or not a property holds for a program entity. The properties and the program entities can differ significantly, but the result of a bit-vector analysis is always a truth value.

Reaching definitions, available expressions, live variables, and very busy expressions form the four most prominent examples of bit-vector analysis because they cover the potential combinations of forward- and backward problems and universally- or existentially-quantified problems respectively. For example, reaching definitions is a existentially-quantified, forward problem, because it detects whether or not there exists a path from a definition to a specific program point. In contrast, an expression is very busy at a program point, if it is used on all paths from a program point to the exit of the method and the other examples capture the remaining cases.

The examples show that the program entities under consideration can be quite different. Reaching definitions analysis is concerned with definitions of variables, which are program points where a variable is assigned a value. Obviously, there can be several different definition points for a single variable. Very busy

expressions and available expressions deal with arithmetic subexpressions and live variable analysis answers the question if there exists a path to the exit of a method on which a specific variable is used.

Even though the program entities under consideration differ the representation of the result is usually modelled by a simple set. Program entities which are in the set exhibit the property and the other program entities do not. The data flow lattice is the power-set lattice which consists of all subsets of the set of all program entities. The order relation of this lattice is the subset relation, i.e. a subset is smaller than its supersets. Thus, the greatest element is the full set which includes all program entities while the empty set is the smallest element of the power-set lattice.

Analyses which operate on the power-set lattice are called bit-vector analysis because bit-vectors can represent sets and set operations very efficiently. To achieve this, a single bit in a bit-vector is associated to a program property. The truth value of this bit determines whether or not the property holds at a program point. From the set-based point of view the bit determines if the program property is in the set or not. Technically, the bit-vector representation is very memory efficient, because a single bit suffices to represent each program property. Furthermore, set intersection and set union boils down to logical AND and logical OR respectively. Conceptually, the bit-vector representation decomposes the monolithic set representation into a bit-representation for each program property. This is remarkable because at this point the relationship to the environment model becomes apparent: We can identify the program entities of the bit-vector problem with data flow variables and use the simplest possible lattice which consists of the extremal elements only as the inducing lattice. If the extremal elements are identified with the truth values `true` and `false`, then the program environment becomes a mapping from data flow variables to truth values, which is directly corresponds to the bit-vector representation.

The simple structure of the power-set lattice directly implies that the structure of the summary function representation stays simple, too. We investigate the effects by the specification of instruction-level summary function for separable and non-separable bit-vector analyses.

7.1.1 Separable Bit-Vector Analyses: Reaching Definitions

We can model the reaching definition problem by a data flow variable for each variable definition in the program. The environment mapping maps each variable to `true` if the definition reaches the program point under consideration and to `false` otherwise. Thus, the *inducing lattice* is the boolean lattice where `false` corresponds to the most optimistic element $\top$ and `true` corresponds to the most pessimistic element $\bot$.

The safe approximation operator in this boolean lattice is the logical OR because a definition needs to reach a specific program point by a single path only.

The simple structure of the inducing lattice yields very simple expression structures in the summary function model. In the traditional formulation of

bit-vector analyses the instruction-level transfer functions are usually specified in terms of so called *GEN*- and *KILL*-sets by the following equation:

$$OUT = (IN \setminus KILL) \cup GEN$$

The intuition is that the data flow information which is valid after the execution of an instruction (*OUT*) can be computed from the information which was valid before the execution of the instruction and not invalidated by the instruction (*IN* \ *KILL*) combined with the information generated by the instruction (*GEN*).

For the reaching definition problem the *GEN*-set of an assignment statement at program point n like `n: x = ...` just contains the definition x_n because the new definition of variable x is available immediately after the instruction. Furthermore, the instruction invalidates all other definitions of x which may have reached point n. Thus, the *KILL*-set contains all definitions which refer to a definition of variable x.

GEN- and *KILL*-sets directly translate to summary functions in the data flow expression model. Elements in the *GEN*-set are always `true` after the instruction, while elements in the *KILL*-set are always `false`. Thus, a summary function which operates on environments maps elements in the *GEN*-set to the constant `true` and elements in the *KILL*-set to the constant `false` respectively. Furthermore, all other elements remain unchanged which is captured by the identity mapping as depicted in Figure 7.1.

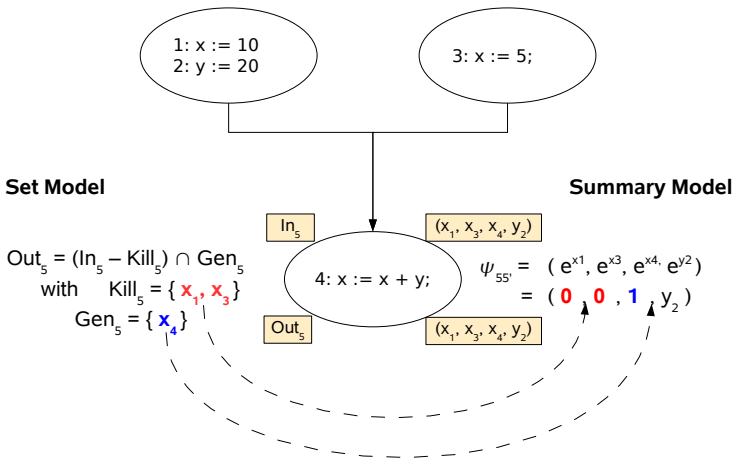


Figure 7.1: Instruction-Level Summary Functions for Bit-Vector Problems

Thus, each defining data flow expression for a definition x_i in an instruction-level summary function is either $\top$, $\bot$, or x_i. Therefore, the summary functions which are computed during the function computation phase remain structurally simple. The reason is that summary function composition and the meet of

summary functions always yield one of three elementary expressions again. For example, the substitution of x_i with one of the elementary expressions yields the substituted expressions while the constants $\bot$ and $\top$ remain unchanged during substitution. Similarly, the meet of two elementary expressions like $x_i \sqcap x_i$ or $x_i \sqcap \bot$ reduces to x_i or $\bot$ according to the normalisation rules for data flow expressions (see Section 5.3.1).

One reason for the efficiency of the summary function representation in this particular case is the simple data flow lattice which consists of the extremal elements of the lattice only. This allows to use the normalisations defined for these special constants. Secondly, the reachability of a definition after an instruction cannot depend on the reachability of other expressions but only on the reachability of itself. Therefore, a defining expression of a data flow fact x_i can only contain the variable x_i and no other variable. This is why the reaching definitions problem is called a *separable* bit-vector analysis because the problem can be solved for each definition independently from the reachability of all other definitions.

As a consequence, we can conclude that the applicable summary functions of separable bit-vector problems stay linear in the size of the program state representation because all defining data flow expressions consist of a single atomic expression only.

Thus, the validation principle directly applies to separable bit-vector analysis and the use of function variable expressions extends the model smoothly to the incremental scenario. The use of function variable expressions is also simpler for separable analysis because a single parameter expression is sufficient.

7.1.2 Non-Separable Bit-Vector Analyses: Faint Variables

A non-separable bit-vector analysis cannot be solved for each program property in isolation. An example for such an analysis are faint variables because the faintness of a variable can depend on the faintness of other variables. A variable x is called faint at a program point n if on all paths from n to the exit node either

- x is not used before it is redefined or
- x is only used to compute *another* faint variable - e.g. y.

Thus, the faint variable analysis can be considered to be an extended version of the live variable analysis which additionally takes the liveliness of target variables of an assignment into account. As a consequence, the definition of the $GEN-$ and $KILL-$sets gets more complex because it has to incorporate potential dependencies on the faintness of other variables:

$$\begin{aligned} GEN_n &= \{x | x \in LHS_n, x \notin RHS_n\} \\ KILL_n &= \{x | x \in RHS_n, y \in LHS_n, y \notin OUT_n\} \cup \{x | x \in USE_n\} \end{aligned}$$

In this definition the sets LHS_n and RHS_n denote the variables which occur on the left hand side and on the right hand site of an assignment while the set USE_n contains variables used in other statements like `print(x)`.

The important point is the first part of the definition of the *KILL*-set which states that the faintness of a variable is invalidated if the variable is used on the right hand side of an assignment *but only if* the target variable of the assignment is not faint after the instruction. Thus, the faintness of variable x can depend on the faintness of another variable. For example, if variable z is faint after the assignment `z = x + y` the faintness of x and y is *not* killed by the assignment.

The translation into the expression model is again straight-forward, if we take into account that dependencies on the input state [1] are modelled by variable expressions which refer to the corresponding data flow variables. Thus, the instruction-level transfer function of the assignment `n: y = x` can be modelled in a first step as

$$\psi_{nn'} = (e^x_{nn'}, e^y_{nn'}, e^z_{nn'}) = (t(x, y), y, z)$$

where the elementary transfer function t is a placeholder for a function which describes how the faintness of x before depends on the faintness of x and y after the assignment. The dependency on y stems from the restricted definition of the *KILL*-set while the dependency on the faintness of x itself arises from the usual definition of the transfer function for backward problems

$$IN_n = (OUT_n - KILL_n) \cup GEN_n$$

which implicitly propagates all values which are not influenced by the *KILL*- and *GEN*-sets.

The first attempt to specify the instruction-level summary functions of the faint variables problem uses some elementary transfer function t, to capture the fact that the faintness of a variable may depend on the faintness of other variables. However, elementary transfer functions introduce nested expressions into the summary function model.

We have chosen the simple example of the faint variable analysis, to discuss two general techniques to reduce the occurrence of elementary transfer function. The first technique is generally applicable and manifests itself in the definition of the $\overset{POUB}{\longrightarrow}$-normalisation rule. Recall that

$$\text{If} \quad [t(p)]|_{[x_i:=\top]} \sqcap c_{old} = c_{new} \sqsubset c_{old} \quad \text{then} \quad t(p) \sqcap c_{old} \overset{POUB}{\longrightarrow} t(p) \sqcap c_{new}$$

and consider the expression $t(x, y)$ with $y = \bot$. Such an expression can occur, if a summary function that states that y is not faint is composed with the summary function of `y := x`. Then,

$$t(x, \bot) \overset{POUB}{\longrightarrow} t(x, \bot) \sqcap \bot \overset{BSC}{\longrightarrow} \bot$$

[1] which is the output solution for backward problems

because $t(x, \bot)|_{[x:=\top]} = t(\top, \bot) = \bot$. Essentially, the evaluation of the elementary function t under the optimistic assumption that x is faint after the instruction yields the result that x is not faint before the instruction, because y is not faint after the instruction. This technique is applicable to all elementary transfer functions no matter how complex their internal semantics are. It works very well for problems like faint variables, because the non-faintness of a single parameter immediately implies that the whole expression will always evaluate to a non-faint result.

The second technique to reduce the occurrence of elementary transfer functions, is to remove them as far as possible from the specification of instruction-level summary functions. This can be achieved for the faint variable problem because the dependency is just the logical AND operation and the logical AND corresponds to the safe approximation operation of the inducing lattice. Thus, we can simply replace the elementary function application expressions by the safe approximation operator $\sqcap$. The instruction-level summary function in the example simplifies to

$$\psi_{nn'} = (e^x_{nn'}, e^y_{nn'} e^z_{nn'}) = (x \sqcap y, z)$$

The expressions are now subject to the duplicate variable removal and do not contain any nested expression anymore. An immediate consequence is that each defining expression has at most as many subexpressions as there are data flow variables. Thus, the summary function representation will be at most quadratic in the size of the environment. However, this upper bound will usually not occur because it implies a situation where the faintness of a variable depends on the faintness of all other variables. As soon as one of the variables in such a large expression is proved not to be faint ($\bot$) the $\xrightarrow{BSC}$-normalisation reduces the whole expression to $\bot$.

This assumption is also supported by empirical evidence from the graph reachability approach of Reps et al.. For problems which do not require elementary functions in the instruction-level specification our model is equivalent to the graph model and it is observed in [RHS95] that the number of incoming edges for a node in the graph representation is bounded by 2 for many interesting problems and practically the number of edges remains linear in the size of the nodes for other problems, too. Therefore, the number of variables in the summary function representation also stays linear in the size of the environment because data flow variables in expressions correspond to incoming graph edges.

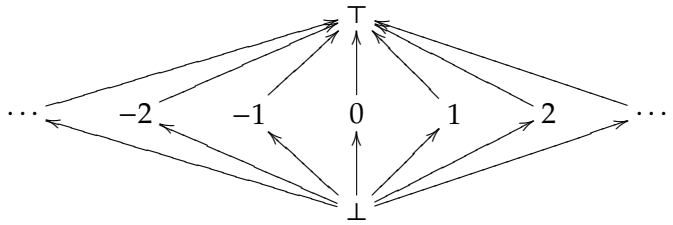
7.2 Constant Propagation

Simple variants of copy constant propagation like copy constant propagation [FL88] and linear constant propagation do not differ much from the bit-vector problems. However, the constant propagation lattice differs from the simple boolean lattice which is the inducing lattice of the bit-vector problems.

Furthermore, the linear constant propagation considers dependencies between data flow variables which cannot be expressed as easily as the dependencies between boolean variables. Therefore, it is interesting to investigate how the summary function model deals with the additional properties of these constant propagation problems.

7.2.1 Arbitrary Lattices: Copy Constant Propagation

Constant propagation does not only compute whether or not a variable contains a constant value but strives to determine the value of the constant. Thus, a simple truth value is not sufficient to represent the data flow information. In contrast, the inducing lattice of constant propagation analyses is augmented with constant values in the following way.



The most pessimistic element $\bot$ states that a variable value is not constant, while the constant elements constitute the fact that a variable has exactly the corresponding value. The conservative approximation operator preserves a constant value, as long as the same constant value is detected on different paths. In contrast, the approximation of two different constant values always yields the most pessimistic element $\bot$.

The most optimistic element $\top$ is an artificial element. It represents "any desired constant" because the safe approximation with any constant value yields the constant value.

The instruction-level summary functions of the constant propagation are fairly simple. Whenever an assignment statement assigns a constant value to a variable, then the summary function generate the appropriate data flow information. Similarly, variable assignments like `x = y` propagate the data flow information from variable y to variable x as depicted in Figure 7.2.

Constant data flow expressions model the generation of data flow information about constants. The dependency on another variables which stems from variable assignments is captured by variable expressions. The loss of data flow information - for example when the program reads an arbitrary value from the input - is expressed by the most pessimistic expression $\bot$. The construction of summary functions combines defining expressions for a data flow variable from different paths on which different variables have been assigned to some variable x a by conservative approximation. Thus, the defining expression of a variable x can contain different variable expressions as subexpressions.

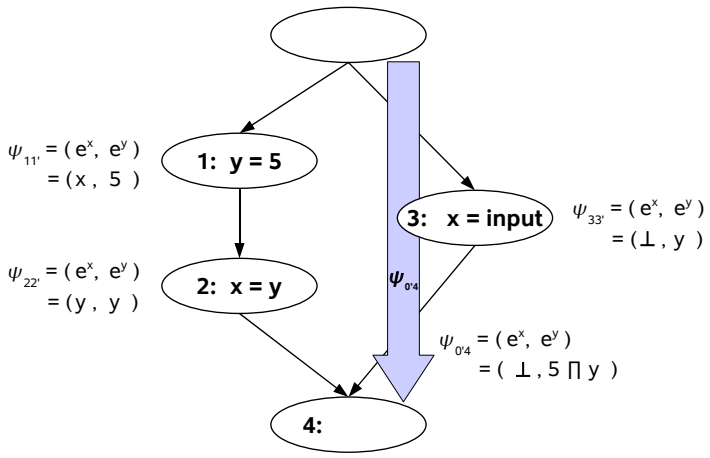


Figure 7.2: Summary Functions for Copy Constant Propagation

All in all, the construction of the instruction-level summary functions for the copy constant propagation problem is similar to the construction for non-separable bit-vector analyses. The difference is only that copy constant propagation uses more expressive data flow values. The constant folding normalisation in the expression model reduces the number of constant expression in each evaluation function to a single element. Therefore, the considerations about complexity of the summary function representation for non-separable bit-vector problems directly apply to copy constant propagation as well: the worst case size of a function representation is quadratic in the size of the environment but the average case is expected to be linear. Once again this statement is justified by the empirical evidence provided in the extended version of the graph reachability approach to interprocedural analysis [SRH96].

7.2.2 Elementary Functions: Linear Constant Propagation

Linear constant propagation is an improvement of copy constant propagation which additionally takes linear dependencies between constants into account. To achieve this, the analysis symbolically executes computations of the form

$$x = a * y + b$$

where x, y are variables and a, b are constant values. Obviously, if y is constant so is x but the *value* of x depends on the linear factor a and b. Linear constant propagation restricts the symbolic execution to linear dependencies because they exhibit some properties which simplify the analysis. The result of all other kinds of arithmetic expressions is still safely approximated by the assumption that the result is not constant.

We will come back to the advantageous properties of linear dependencies at the end of the section. Beforehand, we apply once again the standard specification technique in the expression model that more complex dependencies between variables can be captured in terms of elementary transfer functions.

In order to specify the semantics of a linear arithmetic computation we define elementary functions $l_{(a,b)} : L \to L$ each of which takes a single value y as parameter and maps it to the value $a * y + b$[2]. With such elementary functions at hand, we can immediately define the semantics of the example instruction in terms of an instruction-level summary function as

$$\psi_{ii'}(\dots, x, y, \dots) = \langle e^x_{ii'}, e^y_{ii'}, \dots \rangle = \langle \dots, l_{ab}(y), y, \dots \rangle$$

This model is straight-forward and the instruction-level summaries suffice to perform an interprocedural analysis in the generic framework immediately.

However, the introduction of elementary transfer functions always raises efficiency concerns. Conceptually, the number of elementary transfer functions is not bounded because there is one function for each pair of numbers a and b. Thus, it is possible that the application of summary function composition and meet during the summary computation phase produces safe approximation expressions which contain each potential combination of a linear dependency in the program and a data flow variable from the environment in the program. Thus, the upper bound of the summary function representation raises in a first step to $O(n(nl))$ where l is the number of linear dependencies in the program. The upper bound raises even further, because function composition introduces *nested* expressions, which in turn can contain parameter expressions which have the same complexity as the surrounding safe approximation expression. Even if we bound the nesting depth to a fixed constant, then the worst-case size of the representation has the potential to grow out of control.

Fortunately, this pathological case will usually not occur. Nested expression stem from instruction sequences where one variable *transitively* depends on another, thus

$$\left.\begin{array}{l} i : z = a * y + b \\ j : x = c * z + d \end{array}\right\} \quad \psi_{ij'} = \langle \dots, e^x, \dots \rangle = \langle \dots, l_{cd}(l_{ab}(y)), \dots \rangle$$

Such transitive dependencies can occur in index expressions for multi-dimensional arrays but in such a case the number of dependent variables corresponds to the dimension of the array which is usually quite small. Additionally, the "width" of the expressions increases, whenever a single variable linearly depends on different variables on different paths. We expect such a situation also to be unlikely, and even if it occurs, then its potential effects on the expressions size may very well be limited.

Furthermore, the normalisation of the summary function representation reduces the number of linear dependencies if a non-constant value is detected on one

[2]The functions map lattice values. Thus, they also have to deal with $\top$ and $\bot$ each of which is mapped to itself.

branch ($\xrightarrow{BSC}$-normalisation) of if constants in the input state, yield different constant values ($\xrightarrow{POUB}$-normalisation, followed by a $\xrightarrow{CF}$- and $\xrightarrow{BSC}$-normalisation).

Thus, we can conclude that the specification technique to use elementary transfer functions to express complex data flow dependencies between elements in instruction-level summary function

- is always applicable
- immediately supports the validation scenario
- is for many problems practical, but
- has to accept a potential loss of precision due to safe approximation techniques, which have to be applied to keep the size of function application expressions under control.

At the beginning of the section, we already remarked, that the linear constant propagation exhibits special properties, which ensure that the summary function model can be kept simple. Essentially, all potential linear dependencies can be reduced to the safe approximation of one linear dependency per variable in the environment. Thus, the upper bound for the summary function representation reduces to a size which is quadratic in the size of the environment which models the program state again.

This is exploited in the definition of linear constant propagation in terms of the graph reducibility approach [SRH96] as follows. The key observation is that a transitive linear dependency can be reduced to a direct linear dependency. Consider the example in Figure 7.3. The function composition $l_{(2,7)} \circ l_{(5,1)}$ implies

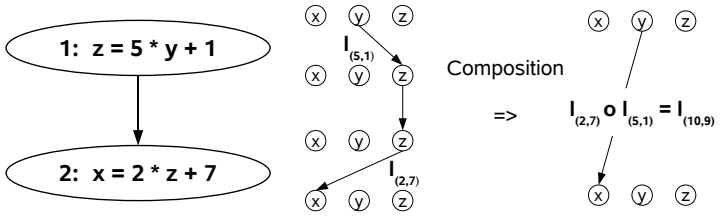


Figure 7.3: Reduction of Transitive Dependencies in the Graph Model

that

$$x = 2 * (5 * y + 1) + 7 = 10 * y + 9$$

and we can model the dependency between y and x directly by $x = l_{(10,9)}(y)$. Essentially, it is easy to compute the *composition* of linear functions. This effectively removes the nesting depth from the representation of elementary summary functions.

However, we already observed that remaining safe approximation expression can still contain an elementary transfer function for each linear dependency in the program and each variable in the environment. Fortunately, the safe approximation of two linear functions which take the same parameter as input, is also computable. Reps, Sagiv and Horwitz [SRH96] choose a representation which exploits the following observation: Two linear dependencies $x = l_{ab}(y)$ and $x = l_{cd}(y)$ between a variable x and a variable y represent two straight lines, so that three cases have to be considered. Firstly, the lines are identical ($a = c \wedge b = d$) then one of the dependencies can be dropped. Secondly, the two lines can be parallel. Thus, the equations are not equal for any y so that the safe approximation of two results can never be a constant value. Therefore, the linear dependencies can be replaced by the most conservative element. Finally, the two lines can intersect in exactly one point, like

$$\begin{aligned} x &= 13y + 3 \\ x &= 11y + 7 \end{aligned}$$

which intersect for $y = 2$ where x becomes 29. To model such a situation, the graph approach represents linear function *internally* by a linear equation and a constant which eventually stores the intersection point. Thus, the safe approximation of the linear dependencies in the example yields

$$l_{(13,3,\top)} \sqcap l_{(11,7,\top)} = l_{(13,3,29)}$$

Observe that the representation of the elementary function has been extended with the third component that models the intersection point.

Thus, all linear dependencies which take the same variable as input can be safely approximated. As a consequence, the upper bound of the summary function representation reduces to $O(n^2)$ because each variable in the environment can produce at most one linear dependency to a single target variable.

However, the reasonable definition of the safe approximation exhibits a subtle problem. It would have been also possible to choose the *second* linear dependency on represent the line part of the representation, i.e.

$$l_{(13,3,\top)} \sqcap l_{(11,7,\top)} = l_{(11,7,29)}$$

Thus, the extended model captures the safe approximation in a reasonable way but there exist several *semantically equivalent* representations for the result of the approximation.

Now, we have reached a very fundamental point which is highly important for the validation of *any* analysis that has to be specified by elementary transfer functions. Assume that the analysis phase derives the representation $l_{(13,3,29)}$

but the validator processes the equation in a different order and comes up with the representation $l_{(11,7,29)}$. Then the validator *cannot* compare these two function representations without knowledge of the *internal* structure of the elementary transfer functions. Essentially, an equality check has to be defined which can detect the semantical equivalence of two elementary function even if their internal structure differs. This is the key difference to the treatment of elementary transfer functions in the expression model, which requires only that each elementary function can be uniquely identified.

To guarantee that the representation of elementary transfer functions [3] stays efficient, the IDE-approach restricts itself to elementary functions which efficiently support the following operations:

- function application - $t(v_1) = v_2$
- function meet - $t_1 \sqcap t_2 = t_3$
- function composition - $t_2 \circ t_1 = t_3$
- equality check - $t_1 = t_2$
- the function lattice has finite height
- closed under function composition and meet

Interestingly, things have come full circle at this point, because these are the central requirements for the interprocedural analysis and validation phase which we already elaborated in Chapter 4! Essentially, the IDE-approach states that the representation of the summary function representation stays efficient if the elementary transfer functions which are used to specify *instruction-level* summary functions have an efficient representation, which can be exploited during function composition and safe approximation of the full-fledged summary functions.

This fundamental observation has several implications for the assessment of the summary function model developed in this thesis:

1. If there is an efficient representation for the elementary transfer functions as postulated by the IDE-approach, then this efficient representation can also be validated if the validator uses the problem-specific operations to check elementary transfer function expressions. Thus, IDE-problems fit into the validation model.
2. Even if an efficient representation of elementary functions does *not* exists, then it is still possible to use the expression model. However, the lack of an efficient representation which is used to compress the representation of elementary transfer functions leads to a significant conceptual increase in the maximum "width" and the nesting depth of expressions. The normalisation rules tackle this growth and they do only require the following operations

[3] Elementary transfer functions are called "value transformer functions" in the original approach.

- function application - $t(v_1) = v_2$
- *identity* check - $t_1 = t_1$

Essentially, reduction from the whole program state to the dependencies between different variables tries to keep the potential influence of elementary functions as local as possible and the normalisation rules try to safely approximate elementary functions by applying them to data flow values inferred for some pieces of the program state during the function computation.

Even though this may not be sufficient from the conceptual point of view, it may still be sufficient from a practical point of view. As we have observed at the beginning of this section, linear constant propagation is an example, where the approach which does not use an efficient representation for the elementary transfer functions can still be practically applicable.

Furthermore, safe approximation techniques which restrict the nesting depth and the width of the expression can keep the expression approach still practical by accepting the inherent loss of precision.

3. The graph model is restricted to problems, for which the dependencies of a single variable on the input state can be decomposed into the safe approximation of a direct dependency for each single variable in the input environment. This means that the model is restricted to unary elementary functions while the expression model can also cope with function expressions that take an arbitrary but fixed number of parameters.

All in all, the reduction of the specification of summary functions to the specification of instruction-level functions which use elementary transfer functions is common to both approaches. However, the expression model unifies elementary functions and other kinds of expressions in a single model. This is necessary, to show that the normalisation rules reduce the defining expressions to a unique normal. This is vital to ensure that the validation process relies on a structural comparison of expression only.

Furthermore, the expression model is flexible enough to cope with problems which do *not* exhibit an efficient representation for elementary summary functions. However, this immediately raises the question if the normalisation rules suffice to keep the resulting size of the summary function representation under control. This thesis does not investigate this aspect further, because its main focus is to consider the *validation* of interprocedural analysis problems.

7.3 Object Oriented Aspects: Type Inference and Call Graph Construction

Type inference is a prerequisite for any interprocedural analysis because the potential receivers of a method call determine the summary functions which describe the semantics of the call. The method name and its signature suffice to

determine the callee in purely procedural languages. Dynamic method binding or function pointers allow multiple candidates for callees at a call site because the runtime environment resolves the call depending on type of the receiver reference or the value of the function pointer.

In order to increase the precision of an interprocedural analysis a static analysis should try to restrict the runtime types of the receiver reference as much as possible. Each potential call target which can safely be ruled out avoids the integration of an additional callee summary at the call site. This is important because any additional callee has the potential to decrease the precision of a subsequent analysis.

The producer can disburden the consumer from the analysis effort, if it ships a safe approximation for the runtime type in the certificate. However, the consumer cannot immediately trust this type information, because faulty and too optimistic type information rules out call targets which can actually be chosen at runtime. As a consequence faulty and too optimistic results of all subsequent analyses can also pass the validation. Thus, the consumer has to validate given type information during the validation process.

To achieve this goal, we formulate the computation of type information in terms of an interprocedural data flow problem and discuss three additional aspects which arise due to the special nature of the type inference problem.

The section is structured accordingly. Firstly, we specify the type inference problem in terms of an interprocedural data flow problem within the summary function model. As usual, this requires the definition of a suitable data flow lattice and the specification of instruction-level summary functions. We use a lattice of type sets in order to improve the precision of the type representation compared to the usual subtype relation. The instruction-level summary functions are closely related to a constant propagation which operates on types and not on integers.

Secondly, we discuss a special challenge of the type inference algorithm: the aim of the type analysis is to provide type information for the construction of the interprocedural flow graph. However, the interprocedural type analysis itself requires an interprocedural flow graph. We can resolve this cyclic dependency either by using safe lower bounds for the receiver references or by an interleaved fix-point computation.

Thirdly, we reinterpret the general validation strategy for the type inference problem. An additional section is dedicated to the question how the restriction to a single software module in the incremental or partial validation scenario influences the validation of type results. Essentially, the validator has to deal with an *open* class hierarchy, because classes which are transmitted to the consumer extend the class hierarchy and thus the type model of the analysis. The type representation of our analysis deals with this problem. Finally, we conclude with a short discussion of other algorithms for call graph construction and investigate whether they are suitable in a validation scenario.

7.3.1 Data Flow Based Type Inference

Our goal is to determine information about the runtime type of the receiver reference in order to restrict the number of potential call targets which have to be taken into account at a dynamically bound call site. Therefore, we define a type inference algorithm which is based on the data flow analysis model presented in Chapter 5. This allows for a validation of the type inference result at the consumer side. The specification of a data flow problem in the summary function model requires the definition of a data flow lattice and the definition of instruction-level transfer function in terms of the summary function model.

Precise Types

Many type analyses use the subtype relation in the class hierarchy to represent types. We adopt and formalise a type model which has already been used as an auxiliary analysis for the definition of method families in partial analysis systems [Thi02]. The integration of the type system in a data flow problem is vital to ensure that the results can be validated according to the general validation principles for interprocedural analyses.

The type model represents types in terms of type sets. Consider the class hierarchy depicted in Figure 7.4 and assume that the analysis determines that

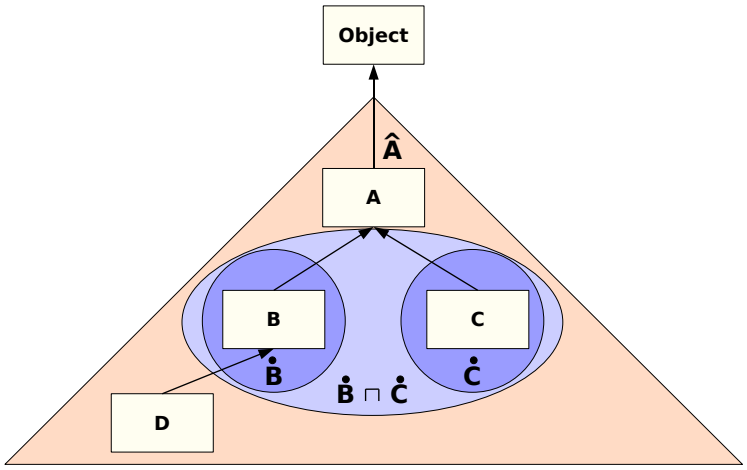


Figure 7.4: Precise Types

the receiver reference of a call is of type B on one path and of type C on another path to a specific call site. If the analysis uses the nearest common supertype of types B and C for the safe approximation at the join point of the two different paths, then this results in type A and all of its subtypes. Now assume that the target method m is declared in each of the classes A, B, and C, then the

analysis has to assume that all three methods $A.m$, $B.m$, and $C.m$ are potential call targets. However, the call will never result in $A.m$ because the assumption that a reference of type A reaches the call site is not true for the two program paths which reach the call site. The imprecision has been introduced by the specification of the safe approximation operator which is more conservative than necessary.

We avoid this problem by representing the type information for a reference by a type *set*. The safe approximation operation is set union and the order relation is the superset relation - i.e. a type set is *weaker* than another type set if it contains *more* types.

This way, the safe approximation of type set $\{\dot{B}\}$ and $\{\dot{C}\}$ yields $\{\dot{B}, \dot{C}\}$ which preserves the information that a reference of type A is not a valid receiver of the call in the example.

A type is a subset of program entities. The type of a reference to an object is a representative for references which target objects of some specific classes. The classes of the program form a class hierarchy. This is captured in the following definitions:

Definition 22 (Class Hierarchy) *A* class hierarchy *is a directed acyclic graph* (C, E) *where* C *is the set of classes in the program and a directed edge* $(c_{sub}, c_{super}) \in E$ *iff* c_{super} *is the immediate supertype of* c_{sub}.

A type B *is a* subtype *of a type* A *if* $A = B$ *or if there exists a path from* B *to* A *in the class hierarchy.*

Obviously, the subtype relation is a transitive and reflexive relation.

Definition 23 (Point Type) *Let* $CH = (C, E)$ *be a class hierarchy. The* point type *of a class* $c \in C$ *denoted by* $\dot{c}$ *represents references to instances of the class* C *and only of class* C.

It is important to observe, that point types do *not* take the subtype relation into account. Thus $\dot{B}$ is a type which represents references to instances of type B only and explicitly rules out references to instances some subclass D of B. This is vital to rule out additional call targets because if some method m is declared in both B and D, then the knowledge, that a reference does not point to an instance of class D, effectively rules out $D.m$ as a potential call target.

However, point types do not fulfil all requirements of a type inference algorithm for separated software modules. Therefore, we introduce the notion of *cone types* as follows:

Definition 24 (Cone Type) *Let* $CH = (C, E)$ *be a class hierarchy. The* cone type *of a class* $c \in C$ *denoted by* $\hat{c}$ *represents references to instances of the class* C *and to instances of all subclasses* C' *of* C.

The term *cone type* emphasises that a cone type represents a whole cone in the class hierarchy while a point type represents a single point (i.e. a node) in the class hierarchy. A cone type captures the "usual" intuition about what a Java programmer considers to be the declared type of a reference. The point types are a more precise representation which enables the type inference algorithm to rule more call targets than by the consideration of the declared type alone.

As stated before, we combine point types and cone types in a *type set* and call such sets *precise types* to emphasise that they represent the classes a reference may point to more precisely than the declared types in the Java program.

Definition 25 (Precise Type) *A* precise type *is a set of cone types and point types. It represents references to all classes which are represented by its point types and cone types.*

Our goal is to specify a type inference algorithm in terms of a data flow problem. Thus, precise types have to form a lattice, which is in fact the case

Corollary 1 *The power-set lattice of the set of all precise types and cone types of a given class hierarchy CH forms a lattice, with respect to the order relation set inclusion and the safe approximation operation set union.*

Proof 16 *Immediate consequence of the fact that a power-set forms a lattice under set union and set inclusion.*

The safe approximation by a union of type sets avoids the potential loss of precision which is typical for the safe approximation which yields the closest common supertype of two given types. Furthermore, the combination of point types and cone types is vital for supporting the analysis of separated software modules but we postpone a more in-depth discussion about the use of cone types to Section 7.3.4 and continue with the specification of the data flow problem at this point.

Instruction-Level Transfer Functions for the Type Inference Problem

The lattice of precise types is able to capture type information more precisely than the immediate use of the subtype relation of the class hierarchy, because a precise type can contain point types. The point type $\dot{A}$ represents a reference which can only point to instances of class A. Such a kind of information is generated only by instructions which create new objects, because a specific class acts as the prototype for the construction of a new object. In Java, objects stay coupled to its "creation class" which was used for its construction throughout their whole lifetime.

Especially, it is possible to determine the call target of a dynamic call *exactly* if we know the creation class of the object the receiver reference points to at runtime. Thus, the proper question a type analysis has to answer is what the potential creation classes of the receiver reference of a call are. In other words,

our type inference analysis tries to follow the data flow from object instantiation sites, where the creation class of an object is known exactly to all the call sites. If it is possible to determine all potential creation sites of a receiver reference of a call exactly, then it is possible to determine all potential callees by simply simulating the method dispatch for each creation class. In the best case, the analysis can determine that all potential object creations use the same class, so that the target method is known exactly.

The instruction-level summary functions which specify such an analysis generate a precise type which contains a single point type for object creation instructions. An assignment instruction copies the type information to the target variable. Essentially, this behaviour is closely related to a copy constant propagation which propagates type values and not integer constants.

Consider the example in Figure 7.5. New objects are created by instruction 1

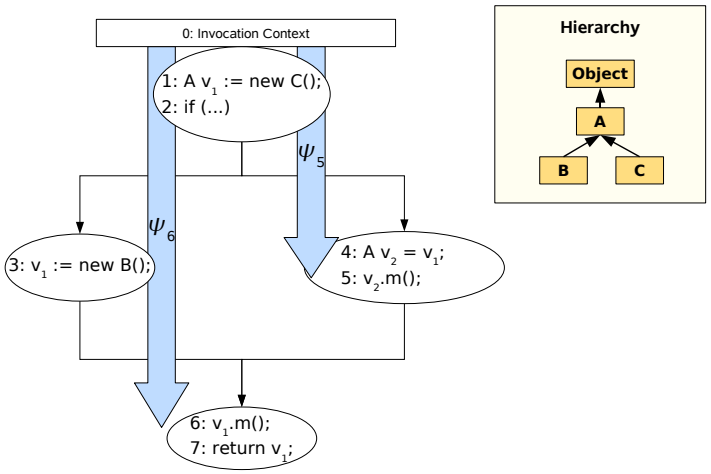


Figure 7.5: Type Inference

and instruction 3 which create instances of class C and B respectively. Thus, the corresponding instruction-level summary functions determine the type information for the target variable v_1 by

$$\begin{aligned} \psi_{11'} &= \langle \ldots, e^{v1}, \ldots \rangle = \langle \ldots, \{\dot{C}\}, \rangle \\ \psi_{33'} &= \langle \ldots, e^{v1}, \ldots \rangle = \langle \ldots, \{\dot{B}\}, \rangle \end{aligned}$$

The semantics of the assignment statement at point 4 is captured by the instruction-level summary

$$\psi_{44'} = \langle \ldots, e^{v2}, \ldots \rangle = \langle \ldots, v_1, \ldots \rangle$$

The generic framework constructs the intraprocedural summary functions from these instruction-level summary functions by applying function composition and function meet.

The summary functions ψ_5 and ψ_6 are of special importance because they map the invocation context to the program state immediately before an invocation of the callee m. This state contains type information for the receiver reference v_2 and v_1 respectively.

For example, the intraprocedural summary function ψ_5 evaluates to

$$\psi_5 = \psi_{44'} \circ \psi_{22'} \circ \psi_{11'} = \langle \ldots, e^{v2}, \ldots \rangle = \langle \ldots, \{\dot{C}\}, \ldots \rangle$$

because the instruction-level summary function of the assignment statement propagates the type information generated by the first instruction from variable v_1 to variable v_2. Thus, the analysis detects, that the receiver references of the call in point 5 will always refer to an object of class C. As a consequence, the dynamic call will always result in an invocation of $C.m$, if method m is defined in class C. This information is more precise than the declared type of variable v_2 because it rules out a method implementation $A.m$.

The fact that the safe approximation of precise types does not loose precision becomes apparent at the join point immediately before instruction 6. The defining expression of variable v_1 in the intraprocedural summary function ψ_6 evaluates to

$$\begin{aligned}
\psi_6 &= ((\psi_{33'}) \sqcap_\Psi (\psi_{55'} \circ \psi_{44'})) \circ \psi_{22'} \circ \psi_{11'}) \\
&= \langle \ldots, \{\dot{B}\}, \ldots \rangle \sqcap_\Psi \langle \ldots, v_1, \ldots \rangle \circ \langle \ldots, \{\dot{C}\}, \ldots \rangle \\
&= \langle \ldots, \{\dot{B}\} \sqcap v_1, \ldots \rangle \circ \langle \ldots, \{\dot{C}\}, \ldots \rangle \\
&= \langle \ldots, \{\dot{B}\} \sqcap \{\dot{C}\}, \ldots \rangle \\
&= \langle \ldots, \{\dot{B}, \dot{C}\}, \ldots \rangle
\end{aligned}$$

The essential observation is the fact that the safe approximation of summary functions $\sqcap_\Psi$ reduces to the safe approximation of lattice elements which in turn is defined by the union of precise types. The type information about the first object creation is propagated via the right execution path, where reference v_1 is not changed. In contrast, new type information about reference v_1 is generated on the left path and the two different pieces of information are joined in point 6.

These quite simple instruction-level summary functions already specify a type inference analysis which is able to track the data flow of reference types through the whole call stack of the program, which contains the local variables. It also includes the parameter passing and return mechanism because the type analysis can use the generic model presented in Section 5.5.

The analysis can determine precise type information, due to the fact that point types represent the creation class of references and that the set-based safe approximation avoids a potential loss of precision at join points.

Class Fields, Object Fields, and Arrays So far, we have not specified the semantics of instructions which access class or object fields, yet. Class fields correspond to global variables and can be represented by additional data flow

variables into the environment which represents the program state as discussed in Section 5.5.1.

As mentioned in the same section, the situation is more difficult for object fields, because there exists a field for each separate object. We observed that there are three different ways, to deal with the situation:

1. The result of each instruction which reads an object field is safely approximated by the generation of the most pessimistic element of the analysis.
2. The analysis can use a single data flow variable for each object field and treat all read and writer operations in a context and flow *insensitive* manner.
3. The analysis can rely on an alias or point to analysis to separate fields in different object instances from each other.

We can now reinterpret these generic strategies in the context of the type inference problem.

The third strategy is the most precise one, but it relies on a *validatable* alias or point-to analysis. Even though it should be possible to specify at least simple variants of this analysis, the framework does not offer an implementation yet. However, the situation reveals a general challenge for the validation scenario: Unlike a normal analysis which simply uses the results of an auxiliary analysis, the validator cannot immediately trust the results of other analyses. Thus, if an analysis depends on another analysis, then the validation of analysis results always requires the validation of all auxiliary analyses as well. Remember that the type inference analysis we discuss here is also an auxiliary analysis for all subsequent interprocedural analysis because it provides the type information which is required to determine the target method of dynamic calls. The validation of an auxiliary analysis is not always trivial as we will see when we discuss the use of type inference results in Section 7.3.4.

The conservative strategy to deal with object fields raises the question what elements of the type lattice safely approximate the type information about references which are read from object fields. The most pessimistic element of the data flow lattice is always a suitable safe lower bound. The order relation of the precise type lattice is set union. Thus, the most pessimistic element is the full set - i.e. the precise type set which contains the point types and cone types of all classes in the class hierarchy. Essentially, the analysis expects that the reference which is read from an object field can refer to object instances of any class in the program.

This safe lower bound is valid, but very conservative. We can improve the safe lower bound of a field access, if we take the *declared* type of the field into account. The static type of a field restricts the references which can be stored in the field, to those which point to an instance of the declared class or one of its subclasses. This intuition is captured by the corresponding cone type, which can act as more precise lower bound for reading field accesses. Thus, the field access `v = a.f` is modelled by the instruction-level summary function $\psi = \langle \ldots, e^v, \ldots \rangle = \langle \ldots, \{\hat{C}\}, \ldots$ if the field f has declared type C. Thus, we use cone types to model safe lower bounds based on the declared type of program

entities. This strategy cannot only be used to deal with object fields but to deal with native methods as well. The safe approximation of a native method uses the most pessimistic summary function to model the effects of the call. Thus, the result type of the method invocation would have been the most pessimistic element of the type lattice. However, even native methods have a declared type whose corresponding cone type can act as a more precise lower bound for the call.

Interestingly, the same principle applies to array accesses as well. The semantics of an instruction which reads a reference from an array, can be modelled by a cone type which corresponds to the declared type of the array elements. Thus, if a reference is read from an array of type A[], then the target variable contains a reference which points to instances of class A or one of its subclasses, which is modelled by the cone type $\hat{A}$

The validator can trust the declared type information because the Java bytecode verifier ensures, that no reference is stored in a field which would violate the type restrictions of the statically declared type. The bytecode verification can also be formulated in terms of a validation problem, which leads to an interesting observation: Sometimes it is possible to specify a simple analysis like the bytecode verification and use its analysis results to increase the precision of a subsequent analysis.

We expect that the safe approximation strategy which takes the declared type of object fields into account is as precise as the strategy which tries to determine the type information in a context and flow insensitive manner for most cases. Therefore, we do not consider the third option to deal with object fields further.

Explicit Casts Up to now we have specified the instruction-level summary functions of object creation instructions, reference assignments, field accesses, array accesses and method invocations. The elements of the inducing lattice and data flow variables suffice to express all of these instructions, if we accept some loss of precision for field and array accesses.

Explicit casts influence the type information in way which can no longer be represented by simple data flow expressions. As usual, we introduce problem specific elementary transfer functions to deal with the issue. Consider the following code snippet:

```
public class B extends A {
  private A fa;

  public void method() {
    fa  = new B();
    B b = (B)fa;
    ...
  }
}
```

A reference to an object of class B is stored in a field of declared type A. If the program reads the reference from the field, then an explicit cast is required before the reference can be assigned to a variable of declared type B. We model the effects of such a cast by an elementary transfer function $ec_B(x)$ which maps a given type to the *set intersection* of the given type and the type of the cast. Thus, the semantics of the cast instruction in the example is modelled by the instruction-level summary function

$$\psi = \langle \ldots, e^{a1}, \ldots \rangle = \langle \ldots, ec_B\{\hat{A}\}, \ldots \rangle$$

The safe lower bound of this expression is the type $\hat{B}$ because the elementary summary function of a cast *improves* the type information, if the parameter type is weaker than the type of the cast.

Additionally, the cast expression preserves more precise type information, e.g. $ec_B(\{\dot{B}\})$ evaluates to $\{\dot{B}\}$ and not to the cone type $\{\hat{B}\}$. This is important to avoid a potential loss of precision by the cast expression. For example, consider the following code snippet:

```
public class A {
  public static A min(A a1, a2) {
    if (a1.isSmallerThan(a2)) {
      return a1;
    } else {
      return a2;
    }
  }
}

public class B extends A {

  public void method() {
    B b1 = new B(3);
    B b2 = new B(7);
    B b3 = (B)A.min(b1, b2);
    ...
  }
}
```

The interprocedural data flow algorithm detects that the result type of the method invocation corresponds to the precise type $\dot{B}$ because the call is treated in a context-sensitive way. However, the programmer has to cast the result to type B in order to meet the semantics of the Java language. If the cast would have been modelled conservatively, then the information that the result type will always be $\dot{B}$ is lost due to the cast. The definition of more precise elementary transfer functions avoids this problem.

7.3.2 Type Inference and Flow Graph Construction

In the preceding section we have specified a type inference analysis in terms of a data flow problem. The goal of this effort is to derive information about the runtime type of the receiver references a dynamically bound method calls. The type information is required to determine all potential target methods of a dynamically bound call, so that an interprocedural analysis can compute the safe approximation of all corresponding summary functions. Essentially, the type information supplies the interprocedural flow graph subsequent analysis operate on. This raises two questions:

1. How do we integrate the type information of a type analysis into the summary function model of a subsequent "client" analysis?
2. How do we resolve the cyclic dependency between the flow graph construction and the type inference analysis which also has to operate on an interprocedural flow graph?

From the point of view of a client analysis the type inference analysis is a module which supplies a type expression for the receiver reference of each call site in the program. The evaluation of this type expression yields a precise type which is a set of cone types and point types. The cone types can also be expanded to all the point types they represent[4]. Thus, the type analysis yields a set of point types, each of which refers to a potential creation class of the receiver reference of the call. Each class exactly defines a single method implementation which is the call target of the dynamic call, if the receiver reference is of that class. If the class C implements the target method m, then $C.m$ is the target of the call. Otherwise, the class has to inherit the implementation from one of its superclasses[5]. Therefore, the look-up mechanism proceeds along the super class chain until it finds the method implementation. This look-up procedure is repeated for each point type which finally determines the set of all potential call targets. A client analysis uses the safe approximation of the callee summaries for all of these call targets as a instruction-level summary function for the call instructions.

Like any of its client analysis, the type inference analysis has to cope with dynamically bound call sites, too. This introduces a cyclic dependency because the type inference analysis also requires a module which determines a safe approximation for the runtime type of the receiver references of dynamic calls.

There are two different ways to deal with this problem. Firstly, the type inference analysis can use a simpler module to compute the type information. For example, the type inference algorithm can use the statically declared type of the receiver reference. We observe in Sections 7.3.1 that the statically declared type is a safe lower bound for the receiver reference. The validator can rely

[4]We postpone the discussion of the impact of missing program parts to Section 7.3.4.

[5]Notice that a creation class cannot be abstract because abstract classes cannot be instantiated. Therefore, each class which is used to create objects has to provide an implementation for all methods of its interface.

on statically declared type because the Java bytecode verifier ensures that only references of the declared type are used as receiver references. The cone type $\hat{A}$ represents all corresponding references and the look-up mechanism for all methods implementations for each point type in the class cone yields a safe approximation for the potential target implementations.

The draw-back of this simple solution is that the type analysis looses precision, because the statically declared type over-approximates the call targets. We can avoid this loss of precision if we interleave the use of the computation of the type information and its use in the following way:

The type inference analysis starts with *most optimistic* assumptions about the type of the receiver references at call sites. The most optimistic element with respect to the order relation of the precise type lattice is the empty type set[6]. The empty type set does not contain any point type and in turn the resolution of the method binding does not yield any call target. Thus, the algorithm inserts the most optimistic summary function at call sites. As a consequence all data flow variables which are modified by the method invocation are set to the most optimistic type result, too. Obviously, the corresponding result of the analysis is too optimistic. However, the analysis weakens the type information about receiver references because precise types which stem from object creations or safely approximated field accesses etc. are propagated to a call site. We use this weaker but more reasonable results in a *second* iteration of the type inference analysis. The weaker results trigger the inclusion of the first callee summaries at call sites and lead again to weaker results. The type analysis continues the iteration until the whole result stabilises.

The result of the interleaved type analysis and flow graph construction is a valid result for the interprocedural type inference problem. It is more precise than the result of the analysis which uses safe lower bounds for the type of receiver references immediately, because the algorithm inserts additional call targets only if a preceding iteration provided the evidence that the receiver reference can in fact point to a specific class. Essentially, the interleaved algorithm computes a *simultaneous* fix-point solution for the type analysis and the flow graph. For a comprehensive discussion of such call graph construction algorithms refer to [Gro98].

7.3.3 Validation of Interprocedural Flow Graphs

In the preceding sections we have specified an interprocedural type inference algorithm which yields safe approximations for the runtime types of receiver references. This information yields an interprocedural flow graph which is a prerequisite for any interprocedural analysis.

It was a central observation that the iterative type inference algorithm computes a simultaneous fix-point solution for the type inference and the flow graph construction problem. Thus, the type inference analysis involves an additional

[6]The order relation is the super set relation, thus greater sets represent weaker results.

fourth fix-point computation next to the tree fix-point iterations for intraprocedural summary function, interprocedural summary functions, and invocation context computation which are inherent to any interprocedural analysis.

Interestingly, the validation of an interprocedural type inference result avoids this fix-point computation, too. A single pass over the program still suffices to validate the whole result. The interprocedural type inference result consists of a safe approximation of the invocation context of each method, intraprocedural summary functions for each flow graph node and an interprocedural summary function for each method, like any other analysis. The checks which ensure that the result constitutes a valid solution for the underlying data flow equation system stay exactly the same.

The only difference to the validation of other analyses concerns the construction of instruction-level summary functions for call instructions. Recall that the validation process constructs the instruction-level summary function of a call instruction in the following way (refer to Section 4.2.5 for details):

$$\psi_{call_m} = \bigsqcap_{i \in target(m)} \psi_{call_{mi}}$$

The instruction-level summary function of a call instruction corresponds to the safe approximation of the interprocedural summary function of each potential callee. We have observed in this section, that the determination of the potential call targets of a dynamically bound call, depends on some safe approximation t of the runtime type of the receiver reference. Thus, the validation relies on the computation of $target(p_{ni}, m_{ni})$ which denotes the set of potential callees of a method call to method m at the call site i in the method n under the assumption given that p_{ni} is a safe approximation of the runtime type of the receiver reference. This computation requires an additional module which yields a *valid* value for p_{ni}.

We can derive this information from the result of the type inference problem by an access to the program state I_{n_i} which is the input state for the call instruction i in method n. This state contains a safe approximation of the type of the receiver reference of the call. It can be constructed from the type inference result by the corresponding intraprocedural summary function ψ_{n_i} because

$$I_{n\_i} = \psi_{n\_i}(IC_n)$$

The validator can compute the safe approximation of the receiver reference during the validation of the type inference result in the very same way. The only difference is that it accesses its *own* analysis while other analyses rely on the result of the type inference algorithm.

The validity of the type analysis results is ensured, because the construction of the instruction-level summary function implicitly injects the assumptions about the receiver reference into the validation process. Essentially, the validator

checks the modified equation system that contains the additional dependencies on receiver types in terms of the augmented determination of the callee targets in $target(p_{ni}, \ldots)$.

7.3.4 Type Inference for Software Modules

The computation of a safe approximation of runtime types for receiver references introduces additional challenges if the validation has to deal with separated software modules.

1. The representation of the data flow results for a software module uses data flow variables and free function variables to capture the effects of other modules which are not yet available. The flow graph construction which depends on the results of a type inference analysis introduces an additional dependency on other modules, because the result of the underlying type analysis can depend on other modules. A straight forward idea to model the impact of open type results on a client analysis is to augment the free function variables with the defining expressions for the receiver reference in the corresponding intraprocedural summary function.
2. The safe approximation principle has to be extended, because not only data flow variables and function variables of the client analysis but also the type expressions of receiver references have to be safely approximated. Essentially, this means, that the validator assumes the existence of some pessimistic method implementation until the defining type expression of the call can be closed.
3. If several software modules are transmitted to the consumer subsequently, then the consumer does not know the complete class hierarchy of the program. An open class hierarchy impacts the determination of the set of potential callees in two different ways. Firstly, if the safe approximation of the receiver reference contains *cone types*, then the validator has to take the existence of additional unknown subclasses into account. Secondly, the determination of the call target for a point type requires the knowledge of the complete super type chain if the corresponding class inherits the method implementation from one of its super classes.
4. One of the advantages of the modular result representation is that it is possible to apply different strategies to close the result of a whole program analysis at the consumer side. We reinterpret these strategies in the presence of open class hierarchies.

This section is structured accordingly and concludes with a brief discussion about optimisation opportunities for the representation of precise types.

Receiver Type Expressions

The determination of all potential callee summaries at a call site depends on a safe approximation of the runtime type of the receiver reference. If the result

of a type inference algorithm is available, then it is possible to derive this type information from the invocation context of the caller and the intraprocedural summary function which maps the invocation context to the call. This technique is applicable immediately if the final analysis result is available.

However, the representation of modular analysis results, uses data flow variables and free function variables to represent the potential impact of other program modules in a flexible way. As a consequence, the result of the type inference analysis can contain function variables because it also has to be represented in a modular way. Thus, it is not possible to *apply* an intraprocedural summary function to derive the invocation context for a call site for two reasons. Firstly, the intraprocedural summary function of the type inference problem can contain free function variables. This happens if the type of a receiver reference depends on the behaviour of a method which is external to the software module under consideration. Secondly, the safe approximation of the invocation context of the caller cannot be trusted before all corresponding call sides have been processed.

Fortunately, the *open* representation of the intraprocedural summary function can be validated and trusted as discussed in Section 5.4. A summary function contains a defining expression for each data flow variable. Especially, it contains a defining expression of the value of the receiver reference of the call. This expression can contain function variables and data flow variables if the runtime type of the receiver reference depends on external methods or on the invocation context, but the expression itself can be validated by an inspection of the caller.

Therefore, we represent each instruction-level summary function of a dynamic call in a client analysis with a type expression given by the module which determines the safe approximation of the type of receiver references at call sites. This type expression is vital to adopt the call target determination for the modular result representation which is discussed in Section 7.3.4.

The extension of function variables with receiver type expressions also impacts the normalisation process. In order to normalise function variable expressions, the normalisation process applies the rule

$$s_{mi}(p_1) \sqcap_L s_{mi}(p_2) \xrightarrow{DSTR} s_{mi}(p_1 \sqcap_L p_2)$$

The intuition of the rule is that two applications of the same summary function on two different program paths are compressed to a single application on a the safe approximation of the parameter environments of the original applications.

Now that we have augmented function variables with type expressions which describe the type of the receiver reference we are confronted with an additional challenge. The function variables stem from two *different* call instructions so that the type expressions of the receiver references can differ. Thus, the normalisation of the original function variable expressions requires the combination of the type expressions. This introduces a subtle challenge, because the original normalisation rule implicitly assumed that a function variable expression

represents the insertion point for exactly one callee summary m. Thus, the normalisation rule just anticipates the the corresponding normalisation

$$m_i(p_1) \sqcap_L m_i(p_2) \rightarrow m_i(p_1 \sqcap_L p_2)$$

which is applied after the substitution of summary function m_i.

The augmentation of the summary function variable with the receiver type expression implies that a single function variable now represents *several* callees each of which belongs to one of the potential call targets of the dynamic call with respect to the given type expression.

Consider the example in Figure 7.6. Two paths which contain two different dynamic calls to a method m. On the left path the type inference analysis has

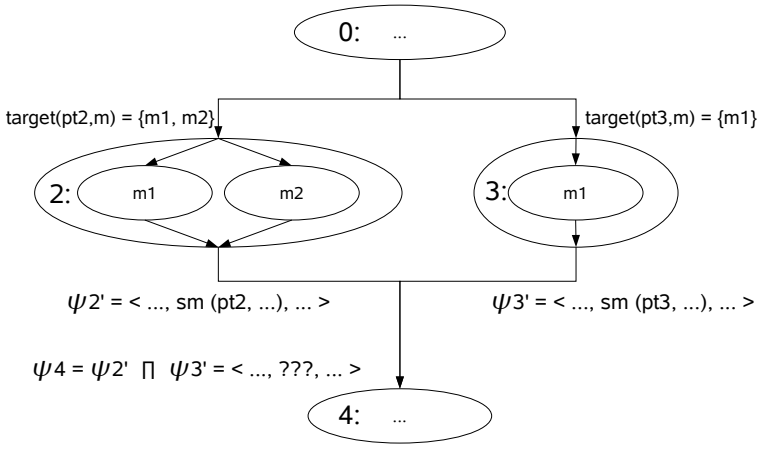


Figure 7.6: Normalisation of Function Variables for Dynamic Calls

determined the precise type pt_2 to be a safe approximation of the receiver type on the left path. This type yields two implementations m_1 and m_2 as potential targets for the dynamic call. In contrast, the type information pt_3 about the receiver type on the right branch is more precise and restricts the set of potential call targets to method implementation m_1.

Thus, the function variable expression $s_m(pt_2, p_2)$ represents the safe approximation of the callee summaries m_1 and m_2 while the function variable expression $s_m(pt_3, p_3)$ represents just the callee summary m_1. If the callee summaries are available than the safe approximation at the joint point, yields

$$\begin{aligned} \psi_4 &= \psi_{2'} \sqcap \psi_{3'} \\ &= \langle \ldots, m_1(p_2) \sqcap m_2(p_2) \sqcap m_1(p_3), \ldots \rangle \\ &= \langle \ldots, m_1(p_2 \sqcap p_3) \sqcap m_2(p_2), \ldots \rangle \end{aligned}$$

The essential observation is that we can combine the two occurrences of m_1 but we have to avoid the introduction of the expression $m_2(p_3)$ because the

type analysis has ruled out the method implementation m_2 on the right branch. Therefore, the corresponding summary function must not be applied to the invocation context of instruction 3. Otherwise, we introduce a potential loss of precision into the normalisation process which in turn can effect the comparability of summary functions which is vital for the validation.

The consequence is that we have to consider the relationship between the receiver type expressions before we apply the $\overset{DSTR}{\longrightarrow}$-normalisation in the extended model for open summary functions. This problem can be solved in two different ways. Firstly, we can restrict the normalisation rule for function variable expressions to expressions which carry the *same* receiver type expression. If the type expressions coincide, then it is ensured that they refer to the same set of call targets so that all call targets occur on both path.

The second option is to decompose the type expressions into one part which is valid on both path and additional type expressions that represent the peculiarities of the specific parts. A function variable expression of the common receiver types can take the safe approximation of the invocation contexts as parameter and the other function variable expressions take the invocation context on the respective path. This way it is at least possible to merge the common part of the input expressions.

We do not consider the challenges of the second approach further because we do not expect the situation to occur often in practice. Thus, the first approach should suffice even though it has the potential to increase the number of subexpressions in the general case. The reason is that a single function variable can occur several times if it is connected to different receiver type expressions.

Safe-Approximation of Type Expressions

The safe approximation of the modular results of a client analysis, requires that the receiver type expression of function variables are safely approximated beforehand. This is necessary because the type expression may be part of a modular result of a type analysis so that it can contain function variable expressions of that analysis.

We observe in Section 7.3.1 that type expressions can be safely approximated in two ways. Firstly, the most pessimistic element of the analysis is always a suitable lower bound. The most pessimistic element of the type analysis is the type set which contains all potential types. As a consequence, the safe approximation mechanism of the client analysis takes all existing method implementations into account, if it uses this type to approximate the receiver type expression of a function variable.

Secondly, we can use the declared type of a program entity as a safe lower bound, because the Java bytecode verification enforces this type. As a consequence, the safe approximation mechanism collects all method implementations in the corresponding cone of class hierarchy and substitutes the function variable with the meet of the summary functions.

Result Determination in the Presence of Open Class Hierarchies

In an open world, the safe-approximation of function variable expressions is further complicated by the fact that the class hierarchy can be expanded with additional classes. The type representation reveals which parts of the type result can be influenced by additional classes because it differentiates between point types and cone types. A point type refers to one specific class in the class hierarchy. Thus, it is *not* influenced by additional subclasses. This is the reason why it is possible to insert all potential callees at a call site if the receiver type consists of point types only even if it is possible to extend the class hierarchy further.

In contrast, a cone type always implies that some additional subclasses have to be considered at a call site. The only safe way to deal with this situation is to assume that some additional class contributes the worst case implementation of a method and to safely approximate the whole call with a safe lower bound. Thus, an analysis significantly looses precision whenever the type analysis fails to restrict the potential type of a receiver reference to a set of point types. Essentially, the safe strategy for the treatment of cone types is an application of the principle which applies "worst-case assumptions" for all external program entities. Additional subclasses are external program entities and the cone types in receiver type expression show how they influence the analysis result.

Additionally, the cone type model can also be used to apply the more optimistic "closed world" and "closed program" assumptions. If the analysis performs a whole program analysis than it assumes that the program will *not* be extended after analysis. Thus, there will be no additional subclasses and the cone types can be reduced to the set of point types of the corresponding cone in the class hierarchy. Similarly, the cone type model can also be used to apply the "closed program assumption" which assumes that the classes of a program are not extended by other software modules but library classes can be extended. As a consequence, cone types which refer to library classes can be treated pessimistically while cone types of program classes can be treated optimistically.

In any case, the cone type model is vital for the representation of the potential effects of dynamic method binding in the presence of an expandable class hierarchy.

Representation of Precise Types

We model a precise type as a set of point types and cone types. Furthermore, we use the power-set lattice of the precise type sets as a data flow lattice. As a consequence, the analysis collects all different pieces of type information which influence a data flow fact at a specific program point in one large type set.

In the worst case such a type set can contain $2 * |C|$ elements if C is the set of classes referenced in the program module under consideration because for each class there exists a point type and a cone type. However, a cone type $\hat{C}$ always

subsumes the corresponding point type $\dot{C}$, so that it suffices to store at most $|C|$ elements.

The representation can be condensed further, if we take the special status of the class `java.lang.Object` into account which is the superclass of all classes and the root of the whole class hierarchy. Therefore, a type set which contains the cone type $\hat{\text{Obj}}\text{ect}$ does not have to store other pieces of type information explicitly, because the cone type of `Object` subsumes all other cone and point types.

These two observations reduce the number of types in a type set significantly for usual programs. The reason is that the class hierarchy is usually very wide because many classes extend the class `Object` but the hierarchy is not very deep, because the specialisation of classes does usually not span many layers of abstraction. Therefore, different pieces of type information either remain in one of the small subtrees, or the cone type of object subsumes the other members of the type set.

It is important, that the representation does not include any knowledge about the super type relation of the different classes. This is vital to ensure that the type representation does not depend on the existence of a complete class hierarchy. As a consequence, the type representation can also be used in an incremental validation scenario, which considers the classes of a program subsequently and builds the class hierarchy step-by-step. Thus, the type sets have to be encoded in the uncompressed set representation

Nevertheless the consumer can use even a partially constructed class hierarchy to further compress its internal type representation. As soon as a class file for a class B has been transmitted, the consumer can extract the immediate super type of the from the class file and integrate it into the class hierarchy. Whenever the partial class hierarchy states that the type B is a subtype of type A, then the cone type $\hat{A}$ subsumes both the point type $\dot{B}$ and the cone type $\hat{B}$.

This way the internal representation of type sets can subsequently reduce to a representation which incorporates the growing knowledge about the class hierarchy at the consumer site. At the same time, the representation stays capable to estimate the potential effects of the unknown parts of the class hierarchy in a flexible way.

7.3.5 Summary and Comparison to Existing Algorithms

The resolution of dynamically bound method calls is a prerequisite for all interprocedural analyses because the potential call targets define the callee summaries which have to be integrated at a dynamic call site. An interprocedural analysis for object-oriented programs cannot afford to treat the influence of dynamic calls conservatively because this would restrict the interprocedural data flow to private and global methods, which are bound statically.

The determination of the potential call target of a dynamically bound call requires type information about the receiver reference of the call and knowledge

about the class hierarchy. The call target of a dynamical method call is defined by the class which was use to construct the object the receiver reference of the call points to. Thus, the type information about the receiver reference should restrict the set of potential creation classes of receiver reference as far as possible.

A simple possibility is to take the declared type of the method into account. The Java bytecode verifier ensures that the receiver reference of a method call points to an object which corresponds to the defining class or to one of its subclasses. Thus, all classes which belong to the cone in the class hierarchy whose root is the declaring class are considered as potential creation classes.

However, this approach requires that the analysis can make some optimistic assumptions. For example, a whole program analysis supposes that the *closed-world assumption* holds - i.e. that the analysis context contains all program entities and that the program cannot be extended. Especially, this implies, that the class hierarchy is fixed and therefore a statically declared type has a fixed set of subclasses.

Unfortunately, the closed-world assumption does not hold for mobile code and the Java environment. It is an essential feature of the Java environment that it provides a mechanism to load program classes via a network connection at runtime. As a consequence, the class hierarchy can evolve during the runtime of a program.

A way to deal with this situation is to make *worst case assumptions* about dynamically loaded classes. The worst-case assumption renders the statically declared type almost useless, because we have to assume that some arbitrary subclass is dynamically loaded and supplies a method implementation for a dynamic call which weakens all assertions about the available implementations. Thus, the summary functions for dynamically bound method calls have to be weakened to the most pessimistic function which rules out any interprocedural data flow which might have been detected during the analysis of the software module under consideration.

An intermediate way is to assume that optimistic assumptions hold about the program in question but that the whole runtime environment can be extended arbitrarily. For example, it can be reasonable to assume that the classes of some specific program cannot be subclassed by dynamically loaded classes because different programs from different sources usually do not know each others code. This *closed-program assumption* allows for an optimistic treatment of the statically declared types which correspond to program classes while all other types are treated according to the worst-case assumption.

However, this approach is only effective after the *whole program* has been transmitted to the consumer site. Therefore, the results cannot be used immediately during the class loading process but only after the arrival of the whole module.

The specification of a data flow-based type inference in this section is a step towards a more precise treatment of the dynamic call resolution. The starting point is the observation that the set of potential creation classes for a receiver reference has to be fixed and that the safe approximation by statically declared

types does not solve this problem without further assumptions if the class hierarchy is expandable. Therefore, we specify a type model which represents an exactly known creation class by a *point type*. Point types originate from object creation instructions and a specialised analysis can investigate the potential data flow from object instantiations to the use of receiver references at dynamic call. The advantage of this analysis is that a fixed set of point types, determines the potential call target *independently* from potential extensions of the class hierarchy, because point types do not implicitly include all subclasses. As a consequence, the dynamic call can be bound to all of its call targets as soon as all classes for the point types are available. This supports an incremental and even a partial validation scenario because it removes the dependency on a completely available class hierarchy.

However, statically declared types are still useful to safely approximate potential effects of other software modules on the type analysis result. These effects arise in an incremental validation scenario if the analysis result depends on the behaviour of code which is not yet available. Furthermore, the current formulation of the type inference problem does not consider all potential data flow in the program. For example, the analysis does not consider the data flow through native methods and via object fields. The statically declared types - or cone types in our terminology - provide a useful lower bound for the potential effects and keep the approximation techniques which rely on some specific assumption like the closed-program assumption still applicable. Thus, the type analysis discussed in this section combines the advantages of a precise data flow based approach with approximation techniques. Furthermore, the results are validatable because the data flow problem has been expressed in terms of the validatable summary function model. Additionally, the discussion reveals, how type inference results have to be incorporated in the validation of the results of a client analysis which uses the type results.

Now, we can investigate the applicability of well-known techniques for the call graph construction for application scenario which involves the validation of data flow results of mobile code.

Class hierarchy analysis (CHA) [DGC95] considers the signature of the method and simulates the method binding strategy in the class hierarchy to find the potential call target. This is the approach which considers the statically declared type of the receiver reference and the corresponding cone in the class hierarchy only. We have already observed that this requires the closed-world or closed-program assumption and prevents the early integration of callee summaries at dynamic call sites in an incremental scenario. The same observation applies to an improved variant of CHA called rapid type analysis [BS96]. This variant restricts the resolution of a dynamic call to those classes which are instantiated within the program. Variable type analysis (VTA) and its variants [BMA03] improve the result even further because it considers which kinds of references are stored in variables during program execution. However, this analysis is flow insensitive and uses a single type value for each variable in the program. This analysis also depends on the closed-world assumption: If the closed-world assumption does not hold then the analysis has to assume that references

to objects with additional types are store into the variables in some of the unavailable pieces of the code. This would significantly reduce the precision of this kind of analyses.

Fragment class analysis [RMR03] does not solve but formalise the problem. The idea is to augment a program fragment with some additional code that exhibits all the potential effects of unavailable code. A whole program analysis of the program fragment of interest and the additional code snippet yields a safe approximation of the analysis result for the fragment. Not surprisingly, CHA and RTA behave poorly in this setting, because CHA has to assume additional subtypes which augment the class hierarchy of the code fragment and RTA has to assume arbitrary instantiation sides. Only data flow based algorithms like Anderson-style points-to analysis [And94] are reported to achieve good results. This is not surprising, because some pieces of the result can solely depend on data flow within the program fragment.

For a comprehensive discussion about various data flow based techniques for the call graph construction refer to [Gro98]. Especially, the thesis discusses several ways to resolve the cyclic dependencies between the type inference algorithm and the construction of the underlying call graph. We have adopted the most precise strategy which uses an interleaved fix-point computation to our application scenario in Section 7.3.3. The original idea for the representation of types in terms of point and cone types to combine the advantages of a data flow analysis and safe approximation techniques stems from an auxiliary analysis for the construction of method families in [Thi02]. The contribution of this thesis is the specification of a type analysis in terms of the summary function model and a comprehensive discussion of its use for the validation of interprocedural analysis results in presence of dynamic method binding and class loading. The main contribution is not the increased precision with respect to a class hierarchy analysis but to ensure the validity of type inference results before the relevant part of the class hierarchy is available.

8 LUPUS - A Framework for Validatable Data Flow Analysis

We have build a prototype implementation of a framework for the computation and validation of interprocedural analysis results - the LUPUS system. The acronym LUPUS stands for **L**ightweight **U**tilisation of **P**rogram Analysis Results from **U**ntrusted **S**ources. The system consists of two components that implement the interprocedural program analysis and the validation of the analysis results as depicted in Figure 8.1.

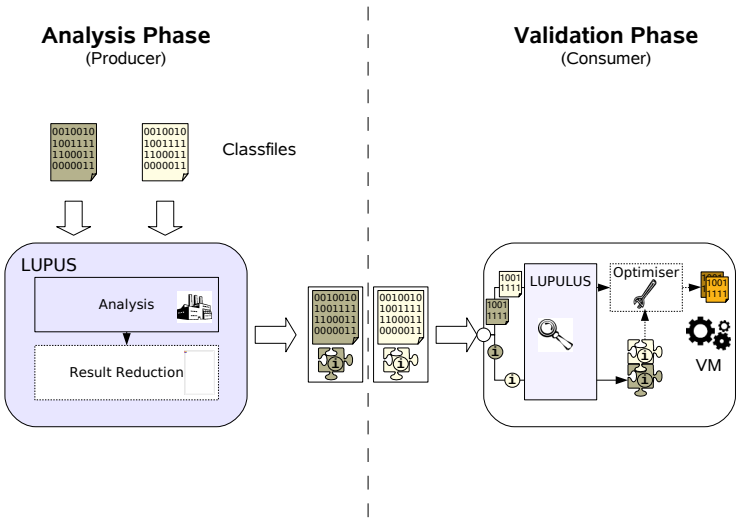


Figure 8.1: Elements of the LUPUS System

The static analysis phase conducts the interprocedural analysis. Furthermore, an additional subsystem can inspect, rearrange, and reduce the precision of analysis results according to the intentional under-approximation strategy discussed in Chapter 6. The implementation of this module requires the development or adoption of demand-driven analysis techniques which is beyond the scope of this thesis. However, an implementation of such a module can easily be integrated into the system later, because the validator can validate any reduced fix-point of an analysis.

The framework expresses the analysis results in terms of the summary function model described in Chapter 5. These results are attached to the class files of the program and shipped to the code consumer. The target platform runs

a lightweight variant of the framework called LUPULUS[1] that performs the validation and composition of the results. After this phase the analysis result are known to be valid. Depending on the application scenario the consumer can safely accept the code or apply optimisations which depend on the results.

Both the analysis and the validation share a common model for the data flow problem in question. They differ in the fact that the fix-point computation phase in the analysis is replaced by a fix-point validation.

The description in this chapter is arranged accordingly. Firstly, we provide an overview of the system architecture before we continue with a discussion of the common base model which is shared by the two phases. Thereafter, we describe the implementation of the analysis and the validation before we conclude with a short comparison to existing frameworks.

8.1 System Overview

The LUPUS framework is structured in three different layers as depicted in Figure 8.2. The *algorithmic layer* contains the analysis and validation algorithms.

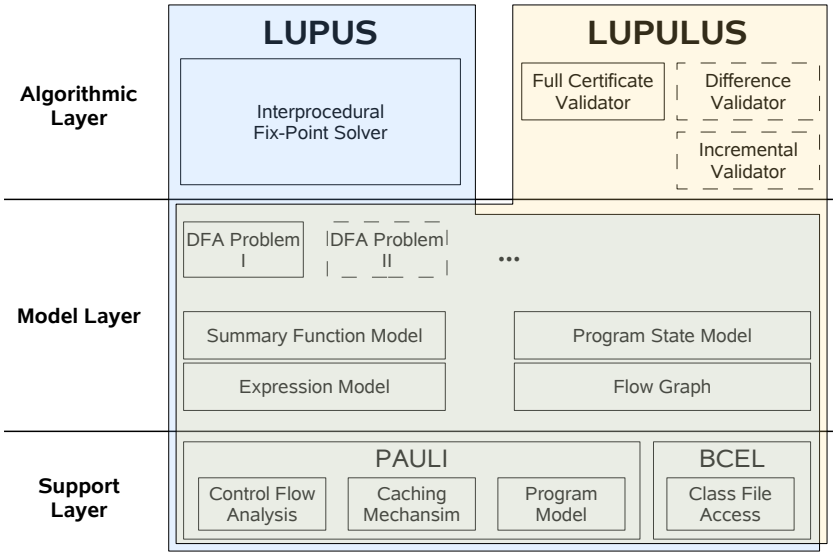


Figure 8.2: System Structure

Currently, the analysis and the validation variant of the framework differ in the algorithmic layer only and share the implementation of the underlying data flow problem in the model layer. The analysis phase is capable to compute open representations for both intraprocedural and interprocedural summary functions.

[1] LUPULUS is the latin diminutive of LUPUS

Furthermore, it is possible to derive the final summary functions for the software module from the open representation by applying different strategies to deal with external dependencies. The validation component currently features an implementation of the full certificate validator only. From the conceptual point of view a difference certificate validator can be implemented relatively easy, because the summary function model supports the determination of difference functions. However, an implementation requires the organisation of the intermediate storage during the validation phase. Essentially, it has to consider the interprocedural dependency model outlined in Section 6.2. Without the capability to drop intermediate result, the difference approach would degenerate to an approach which subsequently constructs a full certificate.

The implementation of an incremental validator is even more challenging, because it requires a careful organisation of the open and applicable summary function representations. The current implementation of the full certificate approach is capable to validate the open summary function representation for intraprocedural and interprocedural summary functions computed by the analysis phase. However, this is only the first step in an incremental validator. It is vital that an incremental validator carefully organises the use of open summary functions and drops this representation as soon as the validity of the corresponding applicable summary function is established. Essentially, this requires the application of the safe-lower bound principle as discussed in Section 5.4. This is challenging from the implementation point of view and only required, in the more advanced incremental and partial analysis scenario. The other application scenario which constructs a modular analysis at the producer site according to the worst-case or closed-program assumption is captured by the prototype already. The draw-back with respect to the incremental validation scenario is that the validation phase has to process the whole result completely.

The *model layer* hosts the key elements of a data flow problem definition: the summary functions, the flow graph, the mapping function, and the data flow lattice. New data flow analyses have to be specified in terms of the summary function model which reduces the specification effort to the specification of instruction-level summary functions. The summary function model plays an outstanding role because it deals with the interprocedural aspect of the data flow analyses. The summary function representation depends on data flow expressions as described in Chapter 5. The main focus of this thesis is to investigate the expressiveness and complexity of the summary function model in different application scenarios. Therefore, the generic model currently uses an implementation which is tailored to expandability. This simplifies the specification and investigation of new analysis problems but raises the problem that the analysis and validation component use a heavyweight infrastructure. This is not a major problem for the analysis phase because we expect that sufficient computational resources are always available at the producer side. However, the situation is not convenient for the use of the framework on a limited device. The severe memory and runtime constraints on a limited device call for a more efficient implementation of the generic function model. A more efficient model would immediately improve the efficiency of the analysis *and*

the validation. However, such improvements raise the question whether the interface for a user can be kept as simple as it is today. An interesting idea is to use a generator which produces adapter code which couples the high-level specification of instruction-level summary functions for a concrete analysis problem with an efficient summary function implementation automatically. However, the construction of an industrial-strength framework is way beyond the aim of this thesis.

An additional *support layer* provides auxiliary services for the model layer. The program analysis framework PAULI[2] supplies basic analyses like a control flow analysis, which is used to build intraprocedural flow graphs of the program, and an abstract model of the program under investigation. The program model is closely related to the abstract syntax tree of the program. The model structures the program into packages, classes, methods and bytecode instructions. The construction of the program model depends on the "Bytecode Engineering Library" BCEL which is capable to process Java class files. It is an important advantage that the whole framework can analyse Java Bytecode, because this allows for an analysis of software components for which the source code is not available. Furthermore, it simplifies the analysis, because it does not have to cope with tasks like name-analysis for local variables or genericity because many of such source-level concepts are explicit in the bytecode.

As an additional service, the framework PAULI offers a plugin mechanism for program analyses and automatic caching of analysis results. This caching mechanism enables the analysis of large software systems, because analysis results are computed on demand, reused as long as they are in the cache, and recomputed automatically if they have been dropped to reclaim memory.

8.2 Implementation of Data Flow Problems

In this section we briefly review the data structures which represent the central elements of data flow problems, how they are implemented in the LUPUS framework, and how they are used to specify new analyses. Section 8.3 and Section 8.4 show how the model specification is used by the analysis framework and the validator respectively.

Generality and expandability are the central design goals of the framework. To simplify the specification of new analysis problems and the integration of other implementation of elements of the basic infrastructure, each component of the analysis model consists of three different parts.

- An interface specifies the high-level view of the component. Other parts of the system usually use this interface to access the component if they do not depend on each other. The interfaces separate the components from each other, so that even central components like the intraprocedural control flow graph can still be replaced.

[2]The framework is developed by the research group "Programming Languages and Compilers" of the University Paderborn

- The framework provides a default implementation for each component which already fixes a large amount of the design decisions. Furthermore, the framework supports the setup of the infrastructure so that the implementor can focus on the specification of the concrete analysis problem as long as the default implementations suffice to deal with the corresponding aspects of the problem. Furthermore, this facilitates the reuse of components in different analysis.
- The user of the framework has to supply an implementation of the components whenever the default implementation does not fit exactly. Some components like the instruction-level summary functions are designed for being extended, so that the user can reuse at least parts of the default implementation.

The following subsections describe the general interfaces of the components, highlight interesting properties of their default implementations, and discuss how a user can specify a concrete analysis problem in the framework.

8.2.1 Elements of a Data Flow Problem

According to Section 3.1 an *analysis problem* consists of four different parts: the flow graph G, a mapping function $[\![]\!]$, a data flow lattice L, and a function space of transfer functions F. The data flow lattice L and the transfer functions in F are independent from the program and form the *analysis framework*[3]. The flow graph and the label function which maps flow graph nodes to their corresponding transfer function in F depend on the program which is subject to the analysis. Therefore, they have to be constructed for a specific program.

Conceptually, the functional approach to interprocedural analysis instantiates the general model in two ways. Firstly, the interprocedural summary computation operates on an interprocedural flow graph and determines a summary function for each program point. Secondly, the value computation phase uses the summary functions as transfer functions to compute a safe approximation of the invocation context of each method. Thus, the summary functions serve two different purposes: They act as data flow values in the first phase and as transfer functions of the second phase. In order to support both tasks any summary function model has to support function composition, function meet, and function application operations. The summary function model in the LUPUS framework defines these operations in a generic way (see Chapter 5).

The interprocedural computation of summary functions has to deal with two additional aspects which do not need to be considered by pure intraprocedural analysis. Firstly, the dynamic method binding has to be resolved at each call site to determine all potential call targets. Secondly, the parameter passing mechanism has to be modelled because the caller and the callee operate both on their own set of local variables The LUPUS framework offers default implementations for these two modules but they can be exchanged if need be.

[3]This is the traditional terminology and must not be confused with the software framework which implements the analysis

8.2.2 Specification of a Concrete Analysis

In order to specify a concrete analysis, the user has to supply instruction-level summary functions and the inducing data flow lattice only as depicted in Figure 8.3. The instruction-level summary functions have to be expressed in terms of

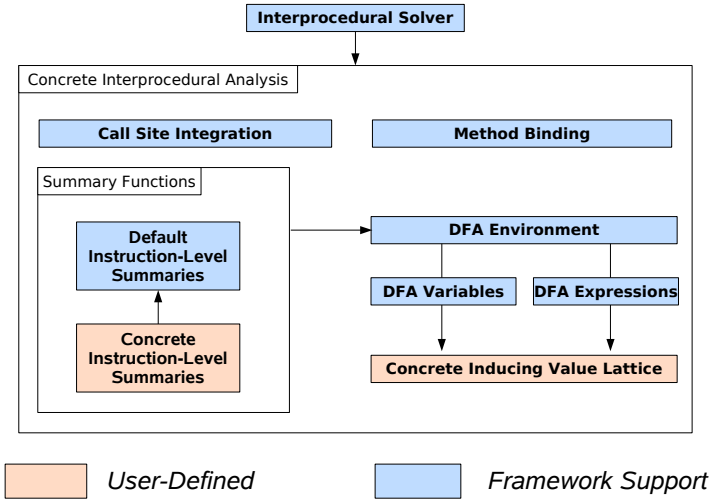


Figure 8.3: Specification of a Concrete Analysis

the summary function model. The summary functions manipulate data flow environments which are mappings from data flow variables to data flow values. The framework offers a default implementation for a data flow environment which supplies mappings for the local variables and the operand stack of the virtual machine. This default environment can be instantiated with arbitrary inducing lattices which supply the data flow values.

Additionally, the framework offers a safe default implementation for each bytecode instruction. The goal is to reduce the number of summary functions which have to be specified by the user to a minimum. The intuition is that the default summary functions specify copy assignments by a corresponding mapping from the source to the target variable and use the most pessimistic element of the inducing lattice wherever data flow information may be generated. Such generation points include object instantiation sites, field accesses and so on. The use of the most pessimistic element of the client analysis safely approximates the behaviour of the corresponding bytecode instruction. The default behaviour deals safely with instructions that are not relevant for the concrete analysis. For example, the specification of a pure integer constant propagation does have to deal with bytecode instructions that operate on references.

After the user has provided instruction-level summary functions and the inducing data flow lattice, the LUPUS framework can set up the analysis automatically. A standard control flow analysis constructs the intraprocedural flow graph. Function composition offers the means to construct the summary function of

each flow graph nodes in the summary function model automatically. The flow node summaries are required for the interprocedural summary computation phase because this phase composes the summary of a flow node with the input summary in order to determine the output summary function.

A separated module is responsible for the integration of summary function at call sites. The default implementation of this module models the parameter passing mechanism by a simultaneous assignment of arguments to parameters. Furthermore, the result value is assigned to the appropriate local variable after the call. This mechanism is appropriate for all analyses which track data flow through local variables, parameters and can be extended to global fields as discussed in Section 5.5.

Finally, the integration of summary functions at call sites requires to deal with dynamically bound method invocations which may target several callees. An additional module determines a safe approximation of all potential callees. The module simulates the dynamic lookup-procedure on an expandable class hierarchy. It determines the currently reachable callees with respect to a type representation for receiver types in terms of the precise type model developed in Section 7.3. It is possible to determine the targets of the call precisely as long as the type representation contains point types only. Furthermore, it is possible to determine if the call target for a single point type is within the current analysis context or external to it.

In contrast, cone types implicitly represent all subclasses of the root class, too. The question whether or not the analysis has to assume the dynamic loading of subclasses depend on the assumptions permitted by the application scenario. It is always possible to deal with the impact of additional subclasses pessimistically and to assume that *any* cone type can refer to some unknown subclass. This worst-case assumption can be relaxed if the application scenario supports the closed-program assumption - i.e. it is reasonable to assume, that no subclasses for a class specified in the software module can be loaded after the module. This assumption allows to treat cone types of program classes precisely after the class hierarchy fragment of the program has been fully constructed.

The resolution strategy is applicable independently of the underlying type analysis which yields the type representation. However, more precise type information can rule out potential call targets. Especially, the potential impact of the dynamic class loading reduces if cone types can be ruled out because the call targets of point types do not depend on additional classes. The current prototype implementation uses a very simple type analysis which yields a single point type for statically bound method invocations like private methods and static methods and a cone type which corresponds to the statically declared type of the receiver reference for dynamic call sites. Conceptually, this strategy corresponds to a class hierarchy analysis, but the module applies the type information in a modular fashion: external callees can be detected automatically and the cone type model allows for the application of the closed-program assumption or other strategies. Additionally, the interface to the underlying type analysis is very thin. Essentially, the binding resolution just expects a precise type

representation which safely approximates the type of the receiver reference at each call site. Thus, the results of a more sophisticated type inference algorithm like the one which is outlined in Section 7.3 can be integrated as soon as they become available.

8.2.3 Flow Graphs and Program Points

The flow graph model of the LUPUS framework uses two different representations for the intra- and interprocedural control flow.

Firstly, the framework constructs a traditional control flow graph for each method body. The control flow analysis is provided by the PAULI framework which can deal with arbitrary class-files by the use of the BCEL-library [4]. The ability to analyse class files is advantageous, because it allows for the inspection of libraries whose source code is not available. Furthermore, class files form the natural transport format for Java code because a target platform usually runs a virtual machine but not a full fledged Java source code compiler.

The control flow graph representation supports the computation of *intraprocedural summary functions* - these are summary functions which map the invocation context of a method to the intermediate program points within the code of the method. This analysis phase does not split the control flow graph at call sites, like this is usually done in other interprocedural analysis frameworks. In contrast, the analysis adds *function variables* into the data flow expressions which represent the intraprocedural summary functions. The central idea is that open summary functions implicitly encode a compressed form of the interprocedural flow graph and that their validity can be ensured due to the validatable summary function model. Thus, the validator does not have to construct or maintain a separate data structure for the interprocedural flow graph.

The function variables act as insertion points for the summary functions of potential callees in the subsequent analysis phase which computes interprocedural summary functions. Conceptually, the analysis switches from the graph representation to a system of data flow equations. A subtle advantage of this representation is that function variable expressions are subject to normalisation like any other data flow expression. This way, the analysis can determine automatically, which method invocations can influence which pieces of the program state. Function variable expressions may be dropped during function composition of function meet. If a summary function is removed during composition, then some new data flow fact has invalidated the influence of a call on a specific piece of the program state. A function variable can be ruled out by the function meet, if the analysis detects a loss of data flow information on a different path already.

The open representation also allows for a flexible treatment of callees. Especially, we can integrate summaries of methods subsequently and we can apply

[4] The Bytecode Engineering Library BCEL is part of the Apache project and provides access to the internal structure of class files.

the worst-case assumption or the closed-program assumption after the whole software module has been analysed. This way, the open representation of summary functions supports a modular analysis at the producer side. Additionally, the open representation is the basis for the more sophisticated incremental and partial validation scenario where the validator has to integrate pieces of the result subsequently. The current implementation of the analysis is able to derive several variants of the open representation and to approximate the results in different ways. The validator is able to validate such open summary functions, too. However, the use of the open representations in a incremental or partial validation scenario requires additional efforts, which we will discuss briefly in Section 8.4.

Irregular Control Flow The current implementation of the framework makes several simplifying assumptions about the flow of control in the subject software. First of all, the framework does not apply special strategies to restrict the possible entry points of the program module under consideration. This is a challenging task in an expandable object-oriented environment, because most methods are callable from unknown code at the first glance. The closed-program assumption cannot be adopted directly, because methods implementation of the software module which override a method of an external class can be called dynamically in the external code. A great number of potential entry points limits the potential precision of the value computation phase significantly, because the analysis cannot make any assumptions about the program state at an entry point. As a consequence, the invocation contexts of internal methods have to be treated pessimistically, if they directly or transitively depend on the invocation context of an entry point. Thus, the information gain of the final value computation phase is likely to be limited unless a - validatable - strategy for the restriction of potential entry points is integrated into the framework. However, the functional part of the analysis already computes a significant amount of valuable data flow information. We compare this information gain of the functional phases with the potential additional information gain of a subsequent value computation phase in Section 9.2.

The framework treats exceptions conservatively, too. All information about the call stack is lost when the flow of control passes to an exception handler. This is a safe but pessimistic strategy and limits the usefulness of the analysis within exception handlers. However, the handling of exceptional states should not occur very often during the normal execution of the program, so that the impact of the precision loss in exception handlers on the analysis result should be limited.

Furthermore, multi-threading and reflection is not considered by the framework yet. This restriction seems to be acceptable, because we expect limited devices not to use these features of the Java language extensively. Method invocations by reflection can be treated conservatively like the invocation of native methods: such calls simply result in the loss of all information about the heap state of the program.

8.2.4 Data Flow Values, Data Flow Expressions and Environments

The model of the data flow lattice consists of two parts: a common interface which describes the generic properties of all lattices and the implementation of the lattice operations and data flow values for each concrete data flow lattice. For example, the lattice of constant values is required for different variants of constant propagation and a lattice which implicitly encodes the class hierarchy is required for a type inference analyses respectively.

The generic properties of the lattice comprise the order relation and the two distinguished extremal elements $\top$ and $\bot$. The extremal elements play an important role in the analysis framework, because they always exist and it holds that the $\top$-element is safely approximated by and the $\bot$-element safely approximates all elements of the lattice. Thus, the $\top$-element forms the natural optimistic initial element because it can reduce to any potential value. The $\bot$-element is even more important, because it indicates the loss of all valuable information. This element does not have to be stored explicitly, because it is always safe to use the most pessimistic assumption if better data flow information is missing.

Furthermore, the $\bot$-element provides an elegant way to express the safe underapproximation of data flow facts. Whenever a piece of data flow information is not yet available the $\bot$-element can act as a safe substitute of the fact. Based on this assumption, the framework can derive safe assumptions about dependent data flow facts as well.

The inducing data flow lattice naturally gives rise to the lattice of data flow expressions as defined in Chapter 5. The building blocks of data flow expressions are generic and can be shared between different analyses which instantiate the expression model with their own value lattice. The expression model offers the following expression types:

A Constant Value Expression models a value of the inducing lattice as an expression.

A Safe Approximation Expression combines two subexpressions with the safe approximation operator of the inducing lattice

An Elementary Function Application Expression represents the application of an elementary transfer function to a fixed tuple of parameters. Elementary transfer functions model complex dependencies between several data flow values if this is required to specify the analysis in question.

A Data Flow Variable is a placeholder for a single data flow fact in the data flow environment. It acts as an insertion point for data flow expressions and data flow values during function composition and function application.

A Function Variable Expression takes an arbitrary number of subexpressions as parameters. The function variable is a placeholder for a summary function and acts as an insertion point of the summary functions of callees.

Data flow expressions form the corner-stone of the summary function model. The specification of an inducing analysis requires the definition of instruction-level summary functions in terms of the model. Thereafter, the framework constructs interprocedural summary functions in a generic way.

Like any lattice, the expression lattice also contains distinguished extremal elements. The $\top$-expressions represents the empty expression whereas the $\bot$-expression represents the safe approximation of all possible expressions. The framework uses the $\bot$-expression once again to improve the size of summary functions and to safely approximate unknown data flow expressions. The $\xrightarrow{BSC}$-normalisation exploits the property of the $\bot$-expression in the normalisation process. This normalisation replaces the safe approximation of $\bot$ with another subexpression because the $\bot$-expression models the safe approximation implicitly already.

The framework also provides a flexible data structure for the representation of data flow environments. An environment is a mapping from data flow variables to data flow expressions. Its purpose is to lift the manipulation of a single piece of program state which is expressed by a data flow expression to the manipulation of the whole state. The set of data flow variables depends on the granularity of the inducing analysis. For example, the default implementation considers the local variables and the elements of the operand stack in a method frame. Obviously, it is inapt to store the mapping for all data flow variables in a data flow environment explicitly. The largest method frames can consist of more than hundred local variables, while the average method manipulates less than the ten local variables. Therefore, the data structure that models an environment contains a default mapping for all variables which are not explicitly mentioned in the environment. Natural choices for the target of the default mapping are the extremal elements and the identity mapping. The extremal elements act as initial choices and safe approximations while the identity mapping represents that all unmentioned data flow facts are not modified by the corresponding summary function.

8.2.5 Summary Function Implementation

A summary function is a tuple of evaluation functions each of which describes the manipulation of a single element of the data flow environment. The LUPUS-framework models summary functions by an environment mapping which maps each data flow variable to the defining expression of the corresponding evaluation function. This way, the implementation of the data flow environment can be shared between the summary function model and a value computation phase which operates on mappings from data flow variables to data flow values.

For example, the default mapping mechanism is shared as well, so that the internal representation of a summary function does not have to mention the mapping for each data flow variable explicitly. This significantly reduces the memory requirements of summary functions for all analyses considered in this thesis.

The LUPUS-framework offers a default implementation of an analysis specification, that already contains instruction-level transfer functions for all Bytecode instructions (see Section 8.2.2). These default summaries also cope with the fact that the Java Virtual Machine is implemented as a stack machine: the push of operands from local variables onto the stack and the store of the operation result, are modelled as assignments. Interestingly, the composition of summary functions usually removes dependencies on the operand stack completely as shown in the example in Figure 8.4.

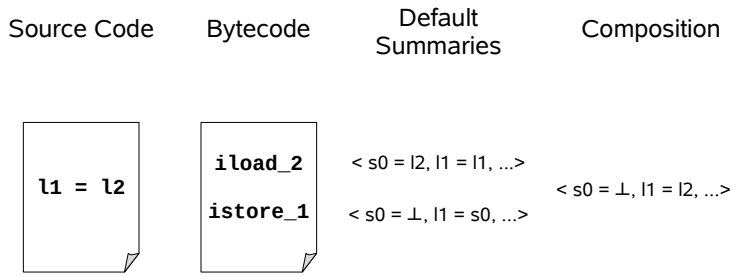


Figure 8.4: Removal of Stack Manipulations by Function Composition

The assignment of the local variables in the source code cannot be performed directly in the stack model: the virtual machine has to load the value of l_2 onto the operand stack before it can be stored into the target variable. This behaviour is resembled by the default summary functions which model data transfer from and to the operand stack. Finally, the composition of these summaries substitutes variable s_0, which removes the indirection that is introduced by the operand stack model of the virtual machine.

The advantage of this straight-forward modelling approach is twofold. Firstly, the framework provides a natural default implementation for the data transfer between operand stack and local variables so that the implementor of a new analysis can focus on the instructions which are relevant for the analysis in question. Secondly, the framework starts directly from Java Bytecode and the validity of the subsequent summary function composition is justified by the properties of the summary function model. Furthermore, the composition mechanism is required anyway, during the computation of interprocedural summary functions. In contrast, frameworks that construct an intermediate representation like three address code impose an additional challenge for the validator: either the validator has to reconstruct the intermediate representation on its own, or it has to validate that the intermediate representation used to

express the analysis results is valid with respect to the given program. The reuse of the summary function model directly at the level of bytecode instructions nicely solves this issue for many analyses in a generic way.

8.3 The Program Analysis Framework

The LUPUS framework implements the summary function approach to interprocedural analysis in three different phases:

1. An intraprocedural analysis computes the intraprocedural summary functions. These summary functions represent the mapping of the invocation context of a method to the program state before or after the execution of each instruction in the method body.
2. The summary functions which result from this phase contain function variables that represent the effects of the callees in the method. An interprocedural analysis computes a fix-point solution for the corresponding system of data flow equations. This solution consists of an interprocedural summary function for each method - i.e. an applicable summary function which maps the invocation context of a method directly to the state upon method return. The final result for the intraprocedural summaries can be computed easily by the substitution of function variables with the final interprocedural summaries.
3. The value computation phase computes the conservative approximation for the invocation contexts of each method. This phase computes a fix-point solution for the dependencies between the final results for the invocation context of the method and the invocation context at all call sites. The application of the final intraprocedural summary functions computes the invocation contexts at call sites from the invocation context of the method, directly. All other intermediate program states within a method can be computed the same way after the result for the invocation context has been established. This phase is currently not fully implemented because it requires a strategy for the restriction of the potential entry points of the module which is non-trivial in an expandable environment. However, we compare the potential information gain of the value computation phase, to the information gain which is already achieved in the functional analysis phase (see Section 9.2).

Two design decisions influence the implementation of the interprocedural analysis significantly. Firstly, the analysis already uses the summary function model presented in Chapter 5. This is advantageous because it enables the reuse of a significant part of the infrastructure in the analysis and the validation phase. However, the analysis "inherits" some of the properties of the model. Most importantly, the implementation currently restricts the nesting depth of defining expressions to a fixed number which can decrease the precision of the analysis. In general, the validation phase does not depend on the implementation of the

analysis phase. An arbitrary analysis framework can be used as long as the analysis results can be specified in terms of the summary function model.

Secondly, the framework does not build an interprocedural flow graph, but operates on the equation systems which is implicitly defined by the summary functions computed in the first phase. We discuss the implementation of the different phase in more detail, now.

8.3.1 Intraprocedural Analysis

The intraprocedural analysis phase computes a summary function for each program point in a method. These summary functions map the invocation context of a method to the program state immediately before or after the execution of each instruction in the method. The summary function which maps the invocation context to the program point after the execution of the return instruction [5] plays a special role because it comprises the interprocedural effects of the execution of a call to the specific method. We call these summary function *interprocedural* summary functions in order to emphasise that they capture the behaviour of a complete method call including all of its subcalls.

The computation of intraprocedural summary functions is a data flow problem which uses function composition with the instruction-level summary functions as transfer functions (see Section 4.2). The instruction-level summary functions correspond to those which would have been used in the intraprocedural counterpart of the analysis. The difference is that the instruction-level transfer functions are now specified in terms of the summary function model, too.

Furthermore, the intraprocedural summary function computation deals with call instructions in a special way. The aim is to compute an open representation of the interprocedural summary function of the method. Therefore, the analysis does not integrate the - potentially unknown - callee summaries directly, but uses function variable expressions to express the effects of a callee summary symbolically. These function variable expressions are first class values in the expression model. Thus, they are subject to normalisation whenever the analysis computes canonical normal forms of the summary functions.

The result of the intraprocedural analysis phase is a single, compact summary function which represents the interprocedural summary functions. It contains function variables for those summaries which influence the final result of the method execution. These function variables can be subsequently substituted by callee summaries as soon as they become available. The interprocedural analysis phase utilises this representation to compute the final interprocedural summaries as discussed in Section 8.3.2.

A special challenge arises from cyclic dependencies *within* the method which lead to an increasing nesting depth of function expressions. Such a situation can arise only if the result of a method invocation contributes to the call context of

[5] If the method has multiple return instructions, then the overall summary is the meet over all return summaries, which corresponds to the introduction of an artificial unique exit node.

the same call in a subsequent iteration of a loop. As a consequence, the function expression which describes the state after the first invocation is substituted into the parameter expressions of the function application expression of the second call during the solution of the data flow problem. Thus, the nesting depth of the function expressions increases on each iteration around the loop. The framework stops this substitution process and determines a function fix-point either by a safe approximation or by the utilisation of special characteristics of the data flow problem in question, as described in Section 5.2.2.

The problem is not a limitation of the fix-point iteration, but our aim to derive a finite representation for the summary function of a method without considering its callees. This is vital to reduce the number of summary functions from one function per control flow node to a single summary function per method and we accept the potential loss of precision to achieve this.

Another idea to construct a finite representation even in the presence of loops is to treat function variable expressions as uninterpreted functions [GTN04]. For example, is possible to represent nested unary uninterpreted function symbols by a string of function symbols. Automata can represent such strings in a finite data structure even if the strings themselves are infinite. The problem is that this theory is not directly applicable to function variable expressions because our functions refer to a tuple of parameter expressions so that the underlying data structure becomes a potentially infinite tree. The current implementation of the framework as well as the underlying summary function model does not support such a kind of representation, yet. Nevertheless, it is an interesting direction of further research to investigate if such modelling techniques are applicable in the validation scenario.

8.3.2 Interprocedural Analysis

The implementation of the interprocedural analysis phase in the LUPUS-framework does not construct an interprocedural flow graph but operates directly on the data flow expression model. The intraprocedural summary function analysis phase computes a function representation which contains function variable expressions for all *relevant* callees. Furthermore, the analysis has already resolved intraprocedural fix-points during the summary function computation.

The goal of the interprocedural summary function analysis is to compute a final interprocedural summary function for each method in the program. This task corresponds to finding a valid substitution for all function variables in the resolvable interprocedural summary functions that result from the intraprocedural phase.

The final summary function results can be computed as follows: Firstly, the analysis module substitutes all function variables in each open summary function by the most optimistic summary function which maps all possible parameter values to the most optimistic element in the inducing value lattice. The resulting

functions are solution candidates for the interprocedural summary function of the corresponding method. Next, an iterative fix-point computation substitutes solution candidates for the function variables in the open representation. This substitution process subsequently weakens the solution candidates. The whole process eventually stabilises as soon as the set of solution candidates forms a valid fix-point solution.

The following considerations explain the algorithmic idea of the implementation. First of all, the system of flow equations which specifies the data flow problem is equivalent to the flow graph model. The propagation of data flow facts in the flow graph directly corresponds to the substitution of data flow variables in the equation system with some values that form the actual solution candidate for the corresponding data flow fact. Therefore, it is more or less a matter of taste whether the algorithm operates on the equation system or on the flow graph representation.

Function variables which refer to external methods complicate the fix-point computation. It is possible to apply the different approximation strategies like the worst-case assumption or the closed-program assumption directly at the call site to remove the function variable expressions of external calls. However, the current implementation computes open representations for the summary functions which still contain external function variables. The advantage is that we can apply the various approximation strategies to the same result representation and that we can evaluate the potential size of open summary function representations as they are required in the incremental or partial analysis. However, the use of function variables for external functions during the computation leads to a potential loss of precision like in the intraprocedural setting, because the conservative limitation of the nesting depth limits the maximum number of unprocessed calls on a path in the intermediate results. However, the situation will improve as soon as more advanced mechanism to represent nested summary functions, which we have outlined in Section 8.3.1, become available.

8.3.3 Solution Analysis and Preparation of the Certificate

The program analysis phase computes the interprocedural result in terms of summary functions and invocation contexts. This result is a valid solution of the equation system which defines the interprocedural data flow problem. The validator can check a complete result easily because it just has to evaluate each right hand side with the values given in the solution and compare it to the defined value.

However, the complete analysis result contains a summary function for each control flow node in the program. Furthermore, a naive encoding of the summary functions can be quadratic in the size of the environment because the mapping of each data flow variable may contain a reference to all data flow variables. The size can even increase further, if the problem specification requires elementary transfer functions. The normalisation rules tackle this

problem already because they reduce the function representation to a canonical normal with a minimal number of expressions. However, the size and the number of the summary functions is still significant.

Therefore, the producer of the analysis results should spend additional efforts to support an efficient validation process at the consumer side. In Section 8.3.3 and in Section 6.4.2 we discuss two techniques which aim at this target. The *lattice strength reduction* technique, reduces the size of the cross-product lattice which defines the environments the validator operates on. This directly reduces the size of the summary functions, because they are linear in the size of the data flow environment. Secondly, the *difference certificate approach* reduces the number and the size of summary functions in the certificate by storing only those pieces of the function representation which differ from the representations which are derived during the validation process anyway. Now, we briefly discuss how these techniques can be integrated into the framework.

Lattice Strength Reduction

The goal of the lattice strength reduction is to reduce the size of the data flow environment, which has to be considered by the validator. The default mapping mechanism in the implementation of data flow environment (see Section 8.2.4) provides the technical support for this technique, because all data flow variables which use the default evaluation functions are not stored explicitly in the environment.

This already reduces the size of the environment in the default implementation of a interprocedural analysis in the LUPUS framework which tracks the data flow through local variables and return values. In such an analysis the environment corresponds to a single method frame. The maximum size of a method frame is bounded by the maximum size of the local variables and of the operand stack which is manipulated by some methods in the analysis context. The maximum size is significant - we encounter methods with more than 70 local variables in the Java 1.5 runtime library. However, the average size of a method frame is very small and involves 5.86 variables on average only. Thus, the use of a default mapping in the environment reduces the memory requirements for the method frame part to less than 10% compared to the straight-forward model. The implementation of the data flow environment adopts itself to the situation automatically, because mappings are only integrated if the corresponding data flow variable is really used.

Admittedly, the same result can be achieved if the construction of the data flow environment for a specific method takes the number of variables into account, which are affected by the method. This is simple for local variables and the operand stack, because the class file contains the maximum number of these variables for each method. Furthermore, the correctness of the numbers is ensured by the bytecode verifier. However, this is not another optimisation technique but the different side of the coin: the validator can either use an adaptive implementation of the environment, or use given knowledge about

the program for the construction of the environments. The second technique requires that the validator protects itself against erroneous values - like the bytecode verifier protects the virtual machine against too small values for the size of the operand stack or the number of local variables. Anyway, both variants aim at the reduction of the lattice strength and this principle can be applied at other points as well.

For example, it is valid to reduce the number of parameter expressions of an unknown call, to those pieces of the program state which are visible to the callee. Essentially, the values of local variables at the call side can be omitted from the argument state of the call, because all arguments are supplied to the callee on the operand stack. Similarly, the state of the local variables and the values on the operand stack of the callee are irrelevant for caller, because these values are invalidated on the call stack upon method return anyway. Thus, it is possible to reduce the strength of the environment of the interprocedural summary function to the mapping of the result value. This has also been observed by Rountev in [RSX08].

The advantage in the validation scenario is that we can extend the lattice strength reduction if it does not depend on language properties but also if it depends on the result. For example, the analyser can ship the information that the data flow via some global fields does not have to be tracked in the certificate. This information can be integrated into the validation process easily, if the validator always assumes that the values which are read from such fields correspond to the most pessimistic value. The adaptive implementation of the data flow environment immediately rules out the corresponding mapping so that only relevant global fields will ever occur in the data flow environment.

Difference Certificate Construction

The idea of the difference certificate approach as discussed in Section 6.1 is to use valid solution candidates which are produced during the validation process directly wherever possible and to ship only difference information if the solution candidate does not match the final result of the analysis.

In the interprocedural setting, the validator has to validate summary functions. Thus, the application of the difference certificate approach requires the determination of *difference functions*, which require a minimal amount of space. Interestingly, the summary function model supports the determination of difference functions directly - we just have to combine the original difference idea for data flow values with the definition of the order relation of summary functions.

The original difference certificate approach exploits the observation that known data flow values can already subsume the unknown ones if the validation process reaches a join point. Thus, the validator can construct the input solution $I^\star$ from the safe approximation $I_c^\star$ of all known input solutions and a difference element Δ which exists only if $I_c^\star$ does not already correspond to $I^\star$ because

$$I^\star = \bigsqcap_{i \in pred} O_i = \bigsqcap_{j \in processedPred} O_j^\star \sqcap \bigsqcap_{k \in unprocessedPred} O_k = I_c^\star \sqcap \Delta$$

We can apply this technique to summary functions immediately if we just integrate the safe approximation over all output summary functions into the certificate if and only if the input solution candidate differs from the final result. However, the safe approximation of all unknown summary functions may be much larger than necessary if it contains the same defining expressions for most variables, because only the differences to the already available candidate are relevant. In Chapter 5 the order relation on summary function representation is specified based on the observation that only a safe approximation expression which contains additional subexpressions is considered to be weaker than a given one. Thus, if the solution candidate $I_c^\star$ does not already subsume the output summaries of the unprocessed predecessor node, then it lacks only some subexpressions in the defining expressions of the whole summary function.

As a consequence, it is possible to determine a much more fine grained representation of a difference function. Essentially, the difference function just has to contain all subexpressions, which are not already present in the solution candidate computed by the validator. Conceptually, we apply the difference computation principle to each defining expression separately and the definition of the order relation on expressions directly yields the missing expressions, which have to be stored in the difference function. This way, the summary function model already supports the core mechanism which is required to prepare a difference certificate after the analysis phase.

8.4 LUPULUS - An Efficient and Flexible Validator

The validator has to check that a given data flow result is valid with respect to the given program. To achieve this, the validator has to check two different kinds of properties: Firstly, the data flow results have to express the semantics of *join points* correctly. This check ensures that the influence of the flow structure of the program is modelled correctly. Secondly, the validator has to check that the *transfer function* of each instruction in the code has been applied correctly. This ensures that the data flow result models the semantics of the code with respect to the data flow problem correctly. Throughout this section we follow the convention of Chapter 4 where we mark values and functions given in the certificate by an asterisk $*$ and the solution candidates produced by the validator by a star $\star$.

A complete interprocedural data flow result consists of a safe approximation of all invocation contexts for each method and a summary function for each node in the control flow graph of the method. The check of the transfer function semantics proves that the given input and output summary function of a flow graph node n are reasonable with respect to the summary function $\psi_{nn'}$ that describes the semantics of the flow node. To ensure this, the composition of the

input summary ψ_{0n} and flow node summary $\psi_{nn'}$ has to be as least as optimistic than the given output summary $\psi_{0n'}$, thus

$$\psi^*_{On'} \sqsubseteq \psi^\star_{On'} = \psi_{nn'} \circ \psi^*_{0n}$$

Thus, the check of the effects of the code and its summary functions is straight forward.

The check of join point semantics is a little bit more complex because there are different kinds of join points which effect the result of an interprocedural analysis problem:

1. An "intraprocedural join point" is a meet of two different control flows of a method. It is the consequence of conditionals or backward edge of a loop. These join points are known from intraprocedural analysis already.
2. The safe approximation of the invocation context of a method constitutes a kind of "call join point" because it merges the program paths of all calls to the method.
3. Dynamic method binding can be considered to be a switch over all potential callees of the method call where the runtime type of the object or the runtime value of a function pointer acts as a guard. After the call, the flow of control from the different potential call target joins again in a "dynamic binding joint point".

Even though all of these join points involve different kinds of information, the validator checks them in essentially the same way. Before we explore this in more depth, we take a closer look at the different kinds of join points.

The intraprocedural join points essentially encode the control flow graph of the method. If a flow node can be reached by different branches, then the validator has to check that the safe approximation of the output summary functions of the predecessor nodes is at least as optimistic as the given input summary function of the join node, thus

$$\psi^*_{0i} \sqsubseteq \psi^\star_{0i} = \bigsqcap_{j \in pred_F i} \psi^\star_{0j'}$$

This effectively ensures that the assertions about the program state at the join point hold at the end of each predecessor node. This check is again equivalent to the same check at join points in the intraprocedural scenario. The only difference is that summary functions and not data flow values are safely approximated and compared.

The other join points occur in the interprocedural scenario only. Firstly, the given invocation context of each method has to safely approximate the invocation context at each call site. The invocation context of a call site can be directly obtained by the corresponding intraprocedural summary function and the invocation context of the caller. Let O_j denote the invocation context at a call site of method m, then

$$IC^*_m \sqsubseteq IC^\star_m = \bigsqcap_{j \in callsites(m)} O^\star_j = \bigsqcap_{j \in callsites(m)} \psi^*_{0j}(IC^*_j)$$

The involved values differ from the check of intraprocedural join points. However, the check once again requires that the validator constructs a solution candidate by a safe approximation of a number of values.

The additional join points which arise from dynamically bound method invocations lead to a similar check. The semantics of each call-instruction i is represented by an additional instruction-level summary function $\psi^*_{ii'}$ in the certificate. This summary function acts as the instruction-level summary function of the call instruction during the validation of the intraprocedural summary functions of the caller because the validator checks that $\psi^*_{0i'} \sqsubseteq \psi^\star_{0i'} = \psi^*_{ii'} \circ \psi^*_{0i}$. The validity of given summary functions of callees requires that the given summary safely approximates all interprocedural summaries of all potential callees at the call site. Let $calltarget(i)$ denote the set of all callees of the call instruction i, then

$$\psi^*_{ii'} \sqsubseteq \psi^\star_{ii'} = \bigsqcap_{j \in calltarget(i)} \psi^*_j$$

where ψ^*_j denotes the interprocedural summary of method j. This summary corresponds to the output summary of the exit node of the method.

Essentially, all join point tests are structurally equivalent, because the validator has to check that the safe approximation of some given values B^*_j is as least as optimistic than some given value A^*, thus

$$A^* \sqsubseteq A^\star = \bigsqcap_{j \in J} B^*_j$$

This check can be performed in two different ways. Firstly, the validator can compute the safe approximation and check the resulting solution candidate against the corresponding entry in the certificate. Secondly, the validator can also decompose the check into a number of subsequent tests, one for each relevant entry in the certificate which contributes to the test:

$$\forall j \in J : A^* \sqsubseteq B^*_j$$

This is possible because the validator does not have to ensure the maximality of the given fix-point. In order to validate the maximality of the fix point the validator has to check the equivalence of A^* and $A^\star$ which requires the computation of the safe approximation.

Any validator has to ensure that both the transfer function and the join point checks hold for all pieces of the analysis result. The validators differ only in the way how they achieve this goal. Before we take a look at different validation strategies, we consider which parts of the framework infrastructure can be reused by the validator.

8.4.1 Reusable Infrastructure

The structure of the checks that have to be performed by the validator can act as a guide to find out which parts of the framework infrastructure can be reused by the validator.

First of all, the checks rely on the function model, because summary function have to be compared to each other, composed with each other, and they have to be applied to invocation contexts, in order to derive the data flow values that describe the invocation contexts of method calls. Thus, the validation process relies on a correct implementation of the summary function model. This model involves data flow environments and evaluation functions which in turn depend on an implementation of the data flow expression model.

Furthermore, the check of the intraprocedural summary function expressed by the equation

$$\psi^*_{On'} \sqsubseteq \psi^\star_{On'} = \psi_{nn'} \circ \psi^*_{0n}$$

requires valid *instruction-level* summary functions $\psi_{nn'}$.

It is not surprising, that the validator depends on the expression model and instruction-level summary functions because these two pieces of information define the data flow problem in question. The expression model establishes the link to the inducing lattice, because constant expressions correspond to the values of the inducing lattice and the evaluation of expressions corresponds to a computation of lattice values. The instruction-level summary functions define the semantics of single instructions with respect to the data flow problem because they specify how the execution of an instruction changes the assertions about the program state the analysis is able to ensure.

Thus, the validation process depends on a correct specification of the concrete data-flow problem. However, the summary functions model is shared between all analysis - just the implementation of the inducing lattice and the instruction-level summary functions can differ from one analysis to another.

The validator depends on other modules in a general way, too. Firstly, it depends on the control-flow graph of the method, because the check

$$\psi^*_{0i} \sqsubseteq \psi^\star_{0i} = \bigsqcap_{j \in pred_F i} \psi_{0j'}$$

requires the determination of the predecessors of node i in the flow graph F. The validator can either perform a control-flow analysis on its own, or it can check the validity of control flow information supplied implicitly in the certificate.

The additional checks which are required in the interprocedural scenario induce additional dependencies. Both the check of the invocation contexts

$$IC^*_m \sqsubseteq IC^\star_m = \bigsqcap_{j \in callsites(m)} O^\star_j = \bigsqcap_{j \in callsites(m)} \psi^*_{0j}(IC^*_j)$$

and the check of the instruction-level transfer functions of call instructions

$$\psi^*_{ii'} \sqsubseteq \psi^\star_{ii'} = \bigsqcap_{j \in calltarget(i)} \psi^*_j$$

depend on the determination of the targets of a dynamically bound method call. The resolution of a dynamically bound method call depends on precise type

information about the receiver reference of the call. The implementation of the call resolution mechanism is capable to cope with expandable class hierarchies. A point type in the precise type representation defines the call target exactly, and the resolution mechanism can check whether the target method is part of the software module under consideration or not. In the latter case the validator inserts the most pessimistic summary function at the call side. However, this situation only arises if the program inherits a method implementation from an unknown superclass. The treatment of cone types depends on the assumptions which can be made about the dynamic class loading. The worst-case assumption expects that virtually any class can be extended by a subsequently loaded class, so that all call sites which depend on cone types have to be treated pessimistically. The closed-program assumptions relaxes this very conservative approach. However, the analysis phase and the validation phase have to use the same implementation of the dynamic call resolution.

Additionally, the validator has to check the validity of the type information used for the resolution. This is simple in the current prototype implementation because we just use the statically declared type of the receiver reference for the resolution dynamically bound method calls and the Java bytecode verifier checks the validity of this type. Nevertheless, we can also integrate more precise type results if they stem from a validatable type inference algorithm at this point.

The final aspect which also has to be considered during the validation is that the summary functions of callees cannot be integrated directly into the intraprocedural summary functions of the caller because they express the manipulations of the program state in terms of the context of the caller. Thus, the validator depends on the module which supplies the call- and return-functions for method calls, too.

To summarise, the validator reuses significant parts of the analysis framework:

- the summary function model including data flow expressions and their normalisation
- the control flow graph of methods
- the definition of the inducing data flow problem including the inducing lattice and instruction-level summary function
- the type model used to resolve dynamically bound method calls
- and the module which specifies the semantics of method invocation and return in terms of call- and return functions.

The validator does *not* reuse the data flow solver, complex strategies for organisation of the worklist of the solver to fasten the fix-point computation etc. Furthermore, it is possible to support the construction of the relevant data structures in the validator by additional information in the certificate as long as the validator can check the given values easily.

All in all, the validator can establish the validity of a given result in a single pass over the certificate and avoids any iteration which may be required in the analysis phase either due to the fix-point computation or due to interdependencies

between different analyses. This linear pass property is the core reason for the efficiency of the validation process. The efficiency can be increased even further, because the validator considers a very small part of the result during each check only, while the analysis phase has to store a large number of intermediate results simultaneously.

8.4.2 Complete Result Validator

The simplest variant of the validator assumes that the analysis phase stores all relevant pieces of information in the certificate. If the complete result is available, then the validator can simply perform all of the checks. The interface to the certificate just requires query methods for the different kinds of information. It is reasonable to assume that such queries can be answered in constant or logarithmic time, if the certificate organises the information in a hash table or if the order in the certificate takes the known access pattern of the validator into account.

The complete validation run requires two checks for each intraprocedural summary function of each flow node in the program. The first check proves its correctness with respect to the instruction-level summary function and the second one proves that the summary is reasonable with respect to summary of the predecessor or successor in the flow graph. The successor and predecessor information of the flow graph can be validated quite easily, because the targets of conditional branches are constant and explicitly encoded in the bytecode conditional bytecode instruction.

Additionally, the instruction-level summary functions of call instructions have to be checked against the summary functions of all callees. This involves a check of the type hierarchy as well, if more sophisticated strategies are used during the static resolution of the dynamic binding.

All in all, the complete result validator is a very simple module. Its additional memory requirements are negligible because they are bounded by the maximum number of intermediate results required for the check of a single equation and some administrative data required for the check of data structures like the flow graph and the class hierarchy.

However, the complete result validator suffers from two major drawbacks. Firstly, the size of the certificate is large, even if technical optimisations of the data structures like lattice-strength reduction and normalisation of the summary functions are applied. The reason is that the certificate holds two summary functions for each control flow node in the program and a summary function is at least linear in the size of the - potentially reduced - representation of the data flow environment lattice.

The second drawback of the simple validator is that the analysis results cannot be used immediately during the validation process because the producer has applied some strategy to deal with external references which removes function variables from the representation. As a consequence, the validator has to

consider the whole analysis result of a software module to establish the validity of pieces of this result.

This thesis already sets the scene for more sophisticated validation scenarios which target these two draw-backs of the full certificate approach. The difference certificate validation strategy targets a reduction of the size of the certificate, while an incremental validator enables the consumer side to use pieces of the analyses results before the whole program has arrived and it facilitates the use of optimistic assumptions about additionally loaded classes, too.

The summary function model already supports the determination of difference summary functions as explained in Section 8.3.3 and the prototype information of the full certificate validator is already able to validate the open results of a modular analysis where the references to external calls have not been removed. Thus, the framework already provides a significant amount of the infrastructure for the implementation of a difference or incremental validator. The remaining challenge is the implementation of an efficient organisation of the intermediate storage during a more sophisticated validation process.

8.5 Summary and Comparison to Existing Frameworks

We briefly summarise the current state of the implementation, before we compare the LUPUS framework to existing ones. Consider Table 8.1 which classifies the capabilities of the LUPUS-framework in four different categories. The categories state which concrete analyses, which kinds of data flow, which resolution strategies for dynamic binding, and which kinds of validators are supported to what degree by the framework. The degree of support ranges from conceptual support in the model, via framework support for the generic parts of the problem up to a full implementation and evaluation of the specific feature.

The summary in the table shows that the conceptual support in the model is quite comprehensive. The model deals with elementary transfer functions which are required to model analyses like linear constant propagation (LCP) and type inference (TINF). It is also possible to express a reimplementation of the interprocedural def-use analysis with copy propagation, that is one of the most important handcrafted interprocedural analysis in the PAULI framework.

The decomposition of the program state into a data flow environment allows for an integration of fields as soon as they can be identified with a data flow variable. This is straight forward for class fields which are identified by their name. Even accesses to fields of the receiver object of a call can be identified in the bytecode by a specialised intraprocedural analysis that can be expressed in the model comparatively simple. All of these extensions introduce a limited number of additional data flow variables into the environment. The support of data flow that considers object fields is the only point where it is questionable whether a straight-forward use of the model will be possible. A straight-forward use of the model would distinguish fields of object which are created at different instantiation sites by additional data flow variables and the size of the

	Model Support	Framework Support	Implementation and Evaluation
Analysis			
- CCP	+	+	+
- TINF	+	+	(+)
- LCP	+	+	-
- DEFUSE	+	+	-
Dataflow			
- Call Stack	+	+	+
- Global Fields	+	+	-
- this.Fields	+	(+)	-
- Object Fields	(+)	-	-
Dynamic Binding			
- Expandable Type System	+	+	+
- CHA-Based Resolution	+	+	+
- Type-Based Resolution	+	(+)	-
Validation			
- Full Certificate	+	+	+
- Difference Certificate	+	+	-
- Incremental	+	(+)	-

Table 8.1: State of the Implementation

environment would likely become unmanageable. From this observation we conclude that the consideration of general data flow via object fields requires an extension of the environment model which takes the result of a points-to or alias analysis into account.

The integration of the dynamic binding is an example how the consideration of a language feature can lead to further extensions of the summary function model: The model - more precisely the representation of a dynamically bound call as the meet of the summary function of the callees - required additions like a type system which is aware of the expandability of the class hierarchy as well as a parametrisation of function variables with the receiver type of a dynamic call. As a consequence, the validation process had to be adopted as well, to ensure that the consumer also checks the validity of the new features. The underlying class hierarchy can be derived safely from the superclass relation and the validity of the statically declared receiver type is ensured by the bytecode verifier of the JVM. This immediately enables a CHA-based resolution of dynamically bound method calls. Furthermore, the results of a type inference analysis which is specified in terms of the model, can also be used to strengthen the precision of the resolution mechanics.

Finally, the model supports different variants of the validation process in the interprocedural setting. The model expresses the summary function computation as a data flow problem so that a simple validator can check the result by the general validation principle. Furthermore, the summary function model allows for the computation of difference functions, which directly enable a difference

certificate validation. Even an incremental approach is already prepared by the introduction of open summary functions.

The implementation effort which is needed to realise the different aspects can be separated into required additions to the framework and the problem-specific parts of the implementation. The implementation support in the framework is quite advanced. An interface for elementary transfer functions is available and they are already considered during the normalisation process which is vital for the analysis and validation phase. This allows for the specification of distributive analyses that are more expressive than simple bit-vector problems. Furthermore, the existence of an adaptive data flow environment, provides the infrastructure for lattice strength reduction techniques.

The framework solves the resolution of dynamic binding automatically. Function variables carry type information about the receiver reference which corresponds to the statically declared type in the simplest case. The dynamic call resolution is performed on a expandable type hierarchy according to the annotated type during both analysis and validation. Thus, the framework is able to incorporate more advanced strategies for the determination of the receiver type smoothly.

The full certificate and the preparation of a difference certificate are supported by the framework. The incremental approach is partially supported. Open summary functions can be validated like applicable ones but additional bookkeeping mechanisms are required to keep track of the relationship between open summary functions and the applicable ones that represent the final result.

We use the basic facilities of each category for the evaluation of the framework. This is achieved by an interprocedural copy constant propagation which considers the data flow in the call stack of the program and a CHA-based resolution of dynamic method binding. The results are validated by a full certificate validator. Furthermore, the analysis phase computes various kinds of open summary functions, so that it is possible to reason about the information gain in the different analysis phases and to consider the impact of strategies which deal with references to external code (see Chapter 9).

We now compare the current state of the implementation of `LUPUS` to other frameworks. There are two important conclusions. Firstly, the `LUPUS` framework is the first framework that supplies generic support for the validation of *interprocedural analysis results*. Secondly, there is no other framework, which combines the support for various kinds of analyses, data flows, object-oriented aspects like dynamic method binding, and validation techniques in such a unifying way.

However, the `LUPUS` framework cannot cope with well-established research and industrial-strength frameworks with respect to the number of supported analysis and with respect to the technical maturity, yet. For example, both the analysis and the validation phase are based on a common infrastructure which is currently tailored for a smooth representation of the analysis model and which includes limited technical optimisations only. Furthermore, the safe but

very conservative treatment of nested function expressions can lead to a loss of analysis precision if the results are compared to results of existing approaches.

The following sections compare the capabilities of the LUPUS framework to existing frameworks in more depth.

8.5.1 SOOT and INDUS

SOOT is a program analysis tool-set developed by the Sable group of the McGill university [VRCG+99]. It provides framework support for intraprocedural data flow analysis which is tailored to set-based analysis. Interestingly, the SOOT tool-set is able to operate on different kinds of intermediate representations. It features the Jimple intermediate language which is a three address code variant that effectively removes the operand stack model of the Java virtual machine. Shimple is an SSA extension of the intermediate representation.

Interprocedural extensions exist also, ranging from the SPARK framework [LH03] for the implementation of points-to analyses and PADDLE which is a BDD-based variant of interprocedural analysis [Lho06].

The basic infrastructure of SOOT is used by other projects like INDUS from the Santos laboratory of the Kansas state university. INDUS supplies an infrastructure for analysis algorithms and data structures. It already contains a fairly rich set of analyses which range from object-flow analysis, escape analysis, and dependency analysis to more specialised monitor and dead-lock analyses. Additionally, the framework hosts a Java program slicer which implements a rich set of slicing variants [RH07].

Though these frameworks are technically mature, they do not provide a common model for the interprocedural analyses. In contrast, they contain several implementations of specialised interprocedural analyses algorithms which all operate on their own set of data structures. This is an inconvenient situation for the question how to validate an analysis result because new validation strategies are required for each algorithm. The fact that several program analyses depend on each other - for example program slicing relies on a control- and data-dependency analysis - complicate the issue further. As the main focus of this thesis is a general investigation of the validation principles in an interprocedural analysis scenario, we did not try to integrate a validation process into a framework that already incorporates such a rich set of subtle dependencies.

8.5.2 PAG

The program analysis generator PAG [AM95] is one of the few frameworks which supplies infrastructure support for interprocedural analysis in a generic way. The frameworks offers generic solvers for both the call-string and the functional approach.

Essentially, the user just has to define the semantics of instruction-level transfer functions and the analysis lattice and the framework performs the iterative fix-point algorithm. The call-string approach is especially well-suited for the PAG approach, because the framework does only have to apply the instruction-level transfer functions given by the specification of the analysis.

The functional variant of the framework is roughly equivalent to the framework suggested by Knoop in [Kno99]. PAG provides default implementations and interfaces for call-functions and return-functionals. This way, the framework is capable to deal with local variables and return values of recursive functions in the functional setting. However, both frameworks do not provide much support for the implementation of summary functions. Essentially, they expect that the user of the framework supplies efficient implementations for the meet and composition of summary functions.

Though this approach is very flexible, because each analysis can supply problem-specific summary function implementations, it also delegates the question how to compare summary functions to the specification of the analysis. However, this is one of the vital questions, which have to be answered to enable the validation of interprocedural analysis results. The summary function model presented in Chapter 5 solves this problem because it ensures that the function representations reduce to a normal form which can be compared to each other by a simple inspection of the structure of the functions. In contrast, frameworks like PAG leave the solution of such problems to the user of the framework.

Although I decided to stick with the functional model to consider the validation of interprocedural analysis results, the wide-spread use of the PAG framework and the featured call-string approach raises an interesting question for further research: Can we come up with a general validation principle for interprocedural analysis results which stem from a call-string analysis? I think that one of the crucial subgoals for such an approach is to capture the intuition of *realisable* path which prevents the propagation of data flow information along infeasible call path in the call-string approach. This issue is avoided in the functional approach, because the summary functions abstract from concrete data flow values when they are integrated into the call sides in each caller.

8.5.3 SafeTSA

SafeTSA [ADvRF01] is a completely different approach to mobile code safety. The general idea is to transform the program into an inherently safe intermediate format. Only valid programs can be encoded in the intermediate format so that a check of the validity boils down to the proof that the program is structurally correct.

The technique has been originally applied to encode the type safety of a program in the intermediate representation and it has been integrated into a just in time compilation framework by Ronne in [vR05] that also provides traditional analysis techniques like constant propagation and common subexpression removal.

Essentially, the SafeTSA representation provides a secure encoding of the SSA representation of a program which simplifies the application of subsequent analysis phases. However, there is no obvious extension to the interprocedural scenario yet.

8.5.4 Code Surfer

CodeSurfer [BGR05] is an industrial strength framework for the analysis of x86-executable code. Recently, Lim has extended the framework by a transformer specification language TSL [LR08]. The primary goal of this language is to specify the semantics of different instruction sets like the x86, the PowerPC, or the SPARC instruction set in a uniform way, so that data flow analyses which are language independent can be easily applied to different instruction sets.

Conceptually, this approach is comparable to the specification of an analysis based on *instruction-level summary functions* in the LUPUS-framework. Both approaches strive to separate the specification effort in two parts: the generic specification of an (interprocedural) analysis and the specification of the semantics of single instructions. As a consequence, some techniques like the reuse of instruction-level summary functions that are equal for different analyses can be found in both frameworks.

The CodeSurfer framework is more mature because it already incorporates a significant number of data flow analyses - even interprocedural ones - which are based on the same principles as the approach of the LUPUS framework. However, the framework targets low-level machine code and does not explicitly deal with object-oriented aspects and questions related to the validation of the analysis results - which are the main challenges targeted in this thesis.

8.5.5 Abstraction Carrying Code

The abstraction carrying code framework by Elvira Albert et al. [APH05] uses a constraint solver to implement data flow analyses based on the abstract interpretation approach. The framework emphasises the applicability of the general validation principle to arbitrary data flow analyses which are expressed as abstract interpretation problems. Furthermore, the constraint solver framework offers already capabilities to deal with an incremental validation scenario.

However, the framework does not deal with interprocedural and object-oriented aspects like dynamic method binding and parameter passing. Like in other frameworks the bulk of the implementation effort is hidden behind the interface of the abstract interpretation framework.

9 Evaluation

The validation of analysis results consists of three elementary phases. Firstly, the producer analyses the subject software, computes the the analysis results, and encodes them in terms of the analysis model presented in Chapter 5. The output of the *analysis phase* are interprocedural summary functions and invocation contexts which are prepared in a way that the validator can check the correctness of the given results easily. These results are transferred to the consumer in the *transmission phase*. Finally, the validator checks and uses the results in the *validation phase*.

This chapter investigates the behaviour of the three phases on different kinds of subject software. The subject software covers various kinds of runtime libraries, small applications, benchmark programs and larger software systems. Section 9.1 describes the software and compares their characteristics.

The following sections cover each phase of the application scenario separately. The evaluation of the analysis phase focuses on the quality of the analysis results. The analysis phase is capable to derive open summary function representations from the intraprocedural context and for an analysis of the whole software module. We investigate and compare these summary functions to determine the potential and the real information gain of the different phases of the interprocedural analysis. Furthermore, the open representation of summary functions allows for a comparison of different strategies to deal with external code. Thus, we can compare the impact of the worst-case, the closed-program, and the closed-world assumption with each other.

The transmission phase is an important cost factor of the validation approach. The size of the annotations increases the overall size of the program and has to remain acceptable. The annotations contain summary functions and safe approximations of invocation contexts. The size of the summary functions is the central challenge, because a single summary function representation can become quadratic in the program state even if it does not contain nested-expressions which increase the potential size even further.

Finally, the complexity of the validation phase is considered. For the full-certificate approach the memory requirements is closely related to the size of the certificate because a full certificate makes all pieces of information directly accessible for checks during the validation phase. Therefore, the size of the full certificate is also an upper bound for the memory requirements of the difference certificate approach. The intermediate storage will never contain more information even if no intermediate result is dropped at all. A first attempt to estimate the runtime efficiency of the validation phase completes the evaluation.

9.1 Evaluation Setting

In this section we briefly summarise the currently available parts of the framework, the characteristics of the analysis, and the subject software considered by the evaluation. The goal of the evaluation is to investigate the impact of several design decisions on the summary function model and to show that the model is applicable in a concrete context.

The prototype implementation of the framework is discussed in depth in Chapter 8. The analysis part of the framework is capable to determine open representations of intraprocedural and interprocedural summary functions. The open representation of intraprocedural summary functions is derived for each method separately and treats all method calls as calls to external methods. The computation of interprocedural summary functions resolves to calls to internal methods of the software module. However, it can still contain function variables, if an interprocedural summary depends on calls to external methods. For example, some unknown subclass can contribute additional call targets.

The final result for intraprocedural summary functions can be constructed from the interprocedural summary functions comparatively easy. It suffices to substitute the function variables in the initial open result with the interprocedural summaries. The result of this substitution phase consists of intraprocedural summary functions which contain function variables for external method invocations only.

The open summary function representation is flexible, because it is possible to apply different strategies to resolve the references to external code. Examples include an application of the worst-case assumption that just safely approximates the dependent parts of the model and the application of the closed-program assumption which optimistically assumes that classes of the software module will not be extended by dynamically loaded classes. The output of this phase is a final representation of the analysis result under the corresponding assumption.

The validation part of the framework features a full certificate validator which is able to validate open and final summary function representations. In order to validate the final summary function representation it is necessary to apply the same approximation technique for external references in the validation phase which had been used in the analysis phase.

The evaluation focuses on the open and final summary representations because they allow for

- the determination potential and achieved information gain in the different phases of the interprocedural analysis. Particularly, we compare the amount of data flow information detected within methods to the final results of the interprocedural summary function computation. Furthermore, the interprocedural summary functions reveal how much data flow information depends on the final value computation phase.
- the investigation of the impact of the different approximation strategies for external code

- a discussion of the certificate size which essentially depends on the size of the function representation
- an estimation of the impact of the normalisation process and encoding strategies on the size of the summary function representation.

These four topics form the goals of the evaluation of the analysis phase and the certificate size. The evaluation concludes with a brief comparison of the runtime requirements for the analysis and validation in the existing prototype implementation.

9.1.1 Evaluated Analysis

Copy Constant Propagation (CCP) A copy constant propagation which includes integer constants and the `null`-reference forms our primary example analysis. The analysis uses the default implementations of the interprocedural analysis modules of the `LUPUS`-framework to show that these modules already suffice to specify useful analyses. The following list summarises the capabilities of the default implementations (for details refer to Chapter 8).

Instruction-Level Summary Functions The default implementation of instruction-level summary functions tracks the data flow through the local variables and the operand stack of the virtual machine. They use copy semantics for load and store instructions which transfer data between the local variables and the operand stack and safely approximate all instructions which can generate new data flow information - like reading field accesses. The only exception to this rule are call instructions where function variable expressions are integrated in the instruction-level summary. This model is tailored to analyses which use a one-to-one relationship between data flow variables and the local variables in the virtual machine. The specification of the concrete copy constant analysis just contributes the instruction-level transfer functions for the instructions which generate new copy constants - namely the `ICONSTx`, `LDC`, and `ACONST_NULL` bytecodes.

Callee Integration The integration of a callee summary models the simultaneous assignment of arguments to parameters and the assignment of the result value to the operand stack slot which takes the result value in the caller. This way, the analysis is able to track the data flow through the whole call stack of the program.

Dynamic Method Binding The resolution of dynamic method calls depends on a safe-approximation of the receiver type reference in terms of the precise type model developed in Section 7.3. Furthermore, a strategy is required which determines whether or not it is possible to load additional external subclasses for a specific cone type. We investigate the impact of the worst-case assumption and the closed-program assumption on the statically declared type of the receiver reference. This setting can be

interpreted as a class hierarchy analysis which is adopted to the modular setting.

All in all the analysis is fairly simple compared to sophisticated interprocedural analyses which target a specific problem. Nevertheless, it forms a suitable starting point to investigate if the generic summary model is usable to specify validatable interprocedural analyses. Furthermore, we have discussed several extensions and improvements for the default implementations like the analysis of static fields and the use of data-flow based type inference results. The evaluation of a manageable analysis setting answers the question which aspects of the analysis influence the precision of the analysis results in which way. Furthermore, we have to expect that the resource-constraints of a target platform do not suffice to use the results of very ambitious analyses even if the consumer applies validation techniques.

Additionally, the ability to validate the data flow through the call stack of the program is already a useful technique if it is applied in a problem specific way. For example, consider the following code snippet

```
class SecurityChecker {

public static native performSecurityCheck();

public static int securityCheck()
    performSecurityCheck();
    return SECURITY_TOKEN;
}

public static void criticalAccess(int securityToken) {
   ...
}
```

and assume that the consumer side wishes to enforce that client code has passed the security check before it invokes the `criticalAccess`-method. This security policy can be enforced, if a copy constant propagation traces only the data flow of a special constant `SECURITY_TOKEN` which is generated in the `securityCheck`-method. If the analysis yields the result that all call sites of the *criticalAccess*-method in a program pass the special constant as an argument, then this implies that the program must have invoked the `securityCheck()`-method beforehand. The analysis results get fairly simple, because only those variables which are used to pass the `SECURITY_TOKEN` around will contain data flow information which differs from the most pessimistic element.

9.1.2 Analysed Software

The evaluation considers different kinds of software which we subdivide into three larger categories. Firstly, we investigate several instances of the Java runtime library. The runtime library is a prerequisite to run Java programs

and it is available for a wide range of platforms. Secondly, we consider applications and benchmarks of the well known Java Spec Benchmark Suite [Cor]. Single software applications form the usual target of traditional whole program analyses. Finally, we include software frameworks into the evaluation. One of the primary design goals of frameworks is expandability and we want to investigate if this affect the characteristics of the achieved results.

Libraries The Java runtime library has changed significantly during the development of the Java language. Newer versions of the standard library which is part of the usual Java runtime environment have continuously been expanded. Nowadays, the library includes more than 10000 classes and more than 100000 methods. Due to its size, it is already a challenge for an analysis framework. We investigate two variants of the standard library: a modern version of the Java 5 library as well as an old 1.3.1 version.

The 1.3.1 version of the library is a good candidate for a combined analysis with the application programs because the core of the language has not changed very much. Therefore, we expect most application programs to be able to run with this version of the library. At the same time the library itself is significantly smaller than the modern versions. Using a small but sufficiently complete version of the library reduces the analysis effort but avoids to introduce a dependency on an extraction technique. Therefore, we investigate the 1.3.1 version of the library to prepare complete program analyses. Additionally, we stripped away the `javax`-packages which are not mandatory for a valid implementation of the runtime environment to decrease the size of the libraries to some 80 %.

During the evolution of the Java environment, several reduced versions of the standard library have been defined which target smaller platforms than a desktop computer. The Java Micro Edition is tailored for devices with restricted computational capabilities like cell phones or PDAs and an even more restricted set of classes forms the runtime library of the Java Card platform. A Java Card is a chip card with a small microprocessor which is used as a subscriber identity module in mobile phones or as an identification card in public health systems.

These restricted versions of the runtime library are of a special importance because they meet the application scenario of this thesis. A runtime environment which supports the full-fledged standard version of Java is likely to support a data flow analyser as well. Thus, validation is a way to speed up the use of prepared results but it is not mandatory. In contrast, limited devices like mobile phones or smard-cards require the validation approach because the analysis is likely to be prohibitively costly if computed from scratch. The following table summarises the characteristics of the different libraries.

Name	# Classes	# Methods	# CFG Nodes	# Invoke Instr.
jdk1.5.0_07	11572 (C) 1595 (I)	107591 1667 (nat)	479277	288012 (dyn) 155250 (stat)
jdk1.5.0_07 -javax	9252 (C) 1301 (I)	85913 1667(nat)	388366	231444 (dyn) 127080 (stat)
jdk1.3.1	4768 (C) 510 (I)	42434 1394 (nat)	199477	113394 (dyn) 61336 (stat)
jdk1.3.1 -javax	3344 (C) 356 (I)	29647 1394 (nat)	142629	76330 (dyn) 46068 (stat)
j2me_cldc-11	85 (C) 13 (I)	1337 88 (nat)	5990	1614 (dyn) 1968 (stat)
java_card-2_2	69 (C) 27 (I)	449 104 (nat)	1925	303 (dyn) 1054 (stat)

The CLDC library contains only 1337 Java methods in 85 classes and the Java Card library version 2.2 even contains 449 Java methods in 69 classes only. Thus, they form an interesting target for the validation scenario while the investigation of larger libraries shows, that the analysis scales at least for simple program analysis like the copy-constant propagation.

The number of statically bound method calls is significant for the large libraries where roughly every third call site is a static call. However, the statically bound method calls even outnumber the dynamically bound ones, for the restricted version of the library. This may be a consequence of the fact that developers avoid the dynamic creation of objects as far as possible on limited target devices and sacrifice a more flexible expandability which is provided by an more object-oriented programming style. The comparatively high number of statically bound calls in the restricted environments, reduces the potential impact of a pessimistic treatment of the runtime type of receiver references. Thus, it is reasonable to start with a dynamic call resolution based on the statically declared type under the closed program assumption.

Furthermore, the comparison supports our claim that interprocedural analysis techniques are interesting for Java programs. The average number of control flow nodes per method is about four and the average control flow node contains one method invocation. Thus, a significant amount of the control flow in the program depends on method invocations while the average method is structurally simple.

Applications and Benchmarks We include two programs from the Java Spec 98 benchmark and two application programs into the evaluation. We intentionally restrict ourselves to the two largest programs in the benchmark suite because most of the benchmarks either depend heavily on the runtime library or are small applications which aim at a test of the IO or runtime behaviour of the subject software. For example, the `db` and `compress` benchmarks include only 34 and 44 methods respectively. Such small applications are not an interesting target for an evaluation which focuses on average properties of summary func-

tions because the data set is too small. Therefore, the evaluation considers the following programs only

jess: the largest program in the Spec benchmark suite

raytrace: a raytracer implementation of the Spec benchmark suite

jedit: version 4.2 of the well known open-source Java text editor.

xmlviewer: a graphical viewer for XML-documents

Large Software Systems and Frameworks We also investigate different parts of our own analysis framework which consists of the bytecode engineering library `BCEL` (version 5.2), the `PAULI` framework which supplies auxiliary analysis, and the basic infrastructure and the `LUPUS` framework for the interprocedural analysis and validation. The frameworks make only limited use of the capabilities of the Java standard libraries. Most importantly, they rely on the elementary data structures in the `java.util` package. Most of the program logic is implemented by the frameworks themselves, so that it is reasonable to analyse them separately and in combination with the old version of the JDK. Furthermore, the frameworks are designed for expandability which makes them potentially harder to analyse than smaller monolithic applications.

9.2 Evaluation of the Analysis Phase

The primary goal of the evaluation of the analysis phase is to investigate the precision and the structure of the analysis results. A comparison of open summary functions enables us to determine how much of the analysis precision stems from the different phases of the interprocedural analysis. Furthermore, we apply several strategies to deal with the impact of external code and investigate how the various strategies influence the precision of the result. The precision is closely related to the structural complexity of the result because less precise results require less memory to be stored. Thus, the comparison also yields insights about the memory requirements of the different analysis phases.

The computation of summary functions in the `LUPUS` framework is depicted in Figure 9.1. A first analysis phase computes an open representation for the intraprocedural summary functions of each method in isolation. The summaries contain function variables for all method invocations within the method, because each call is treated as an external call. In particular, the phase computes an open representation for the *interprocedural* summary function of the method which is equivalent to the intraprocedural output summary function of the exit node. This open summary function encodes a potentially compressed - variant of the interprocedural flow graph of the method. The representation is compressed, because function variables may have been ruled out already by the partial evaluation during the computation and normalisation of the summary functions.

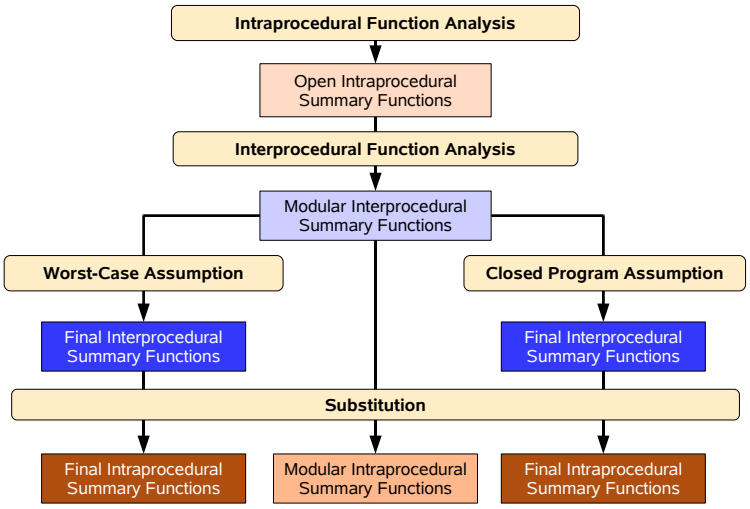


Figure 9.1: Summary Function Computation in the LUPUS Framework

The open representation derived from the intraprocedural context acts as input for the interprocedural function analysis. This analysis phase substitutes function variables which refer to methods within the software module under consideration with the corresponding open summary functions and resolves cyclic dependencies by fix-point iteration. The result of this phase is an open representation for the interprocedural summary function of each method which contains only summary function variables which refer to external method invocations. These open interprocedural summary functions constitute the modular result of the analysis.

The dependencies on external code are modelled explicitly in the interprocedural analysis results. This is useful for the evaluation because it allows to compute the effects of different strategies to deal with external code from the same intermediate result. Function variables remain in the interprocedural summary function representation only if one of its call targets is external. Cone types in the precise type representation of the receiver type are one reason for such a situation because they refer to all subclasses of a class. Other software modules can contribute additional subclasses which in turn can contribute additional call targets for the dynamically bound call. We apply different strategies to deal with this situation. Firstly, the *worst-case assumption* expects that all classes can be subclassed by external code. This strategy treats all function variables which contain cone types pessimistically. Technically, they are replaced by safe lower bounds. The *closed-program* assumption is more optimistic. It assumes that the classes of the analysed software module cannot be extended by additionally loaded classes. This assumption is useful for application programs which depend on an expandable library but which are not designed for expandability themselves. The approximation strategy still treats cone types which originate in a class of the library pessimistically, but "closes" cone types which originate

in a class of the software module. Technically, the cone type is replaced by a set of point types and the function variable is dropped if all corresponding call targets are part of the software module. The safe approximation of open summary functions yields *final* summary functions, which do not contain any function variable anymore.

The interprocedural summary function computation determines a summary function of each method but it does not compute the final intraprocedural summary functions which map the invocation context of the method to each program point within the method. These summary functions can be derived from the open result of the first phase and the interprocedural summary functions computed by the second phase: A subsequent substitution phase replaces all function variables in the intraprocedural result by the interprocedural summary functions computed in the second phase. The result of this substitution can still contain function variables for external methods because the substitution replaces internal calls only. Thus, the remaining function variables have to be treated again with the given safe approximation strategy for external code. The final result of the substitution phase is a final representation of all summary functions. If no safe approximation strategy is applied to external calls, then the result is the open representation of the modular result which still contains the references to external code.

We investigate and compare the structure of the various kinds of summary functions in the following sections. Firstly, we investigate the open representation of the intraprocedural summary functions computed in the intraprocedural phase, because they implicitly encode the result of the intraprocedural counterpart of the analysis. Thus, it is possible to estimate the potential information gain of the subsequent phases and to compare it with the information gain already achieved in the intraprocedural context. Secondly, we investigate the effects of the interprocedural computation phase and the different strategies to deal with external code. The final representations can be compared to the open representation in order to determine how many additional precision could have been achieved by an analysis of a larger context. Thirdly, the applicable representations also implicitly encode how much precision can still be gained by an interprocedural value computation phase.

9.2.1 Intraprocedural Summary Computation

The open summary functions which are computed in the first phase of the summary computation encode the result of a purely intraprocedural analysis. The summary functions contain the dependencies on all callees in terms of function variables. Furthermore, the summary functions contain dependencies on the invocation context in terms of data flow variables.

Thus, the question how much analysis information stems from the intraprocedural context and how much depends on the subsequent interprocedural computation phases boils down to the question how many defining expressions are already constant and how many still depend on function or data flow variables.

All constant data flow expressions are known to be valid independently from the results of the subsequent interprocedural summary function and invocation context computation. Thus, the open summary function representation derived in the first phase allows for a separation of the *intraprocedural information gain* and the *potential interprocedural information gain* just by an inspection of the expression structure within the intraprocedural summary functions. It holds that

Constant Expressions model valuable data flow information which is generated within the method and which does not depend on the invocation context or the callees. This can occur for example in a copy constant propagation if an integer constant is assigned to a local variable.

Most Pessimistic Expressions model the loss of all valuable data flow information due to effects within the method. This can happen for example if two incompatible constants are combined at a join point, or if the analysis makes pessimistic assumptions about values which are read from an object field.

Data Flow Variable Expressions model the fact, that some value depends directly on a piece of the invocation context. Such an expression originates for example from the assignment of a parameter to a local variable.

Function Variable Expressions model the direct dependency of the corresponding piece of the program state to the effects of a single call. Such a dependency is for example generated if the result of the method call is assigned to a local variable of the caller.

Safe Approximation Expressions occur wherever a piece of the program state depends on several pieces of data flow information. Such expressions are created at join points, if the expressions from the different paths cannot be merged by the normalisation process. For example, a constant value on one path may be joined with a data flow variable that describes a direct dependency on a parameter value on another path.

Constant expressions and most pessimistic expressions state what pieces of the program state depend solely on the code in the method. We call these pieces of the result the intraprocedural information gain, because they cannot be influenced by the subsequent interprocedural analysis phase. In contrast, all other expressions have the potential to yield more precise results. The primary goal of the evaluation of the intraprocedural summary functions is to compare the intraprocedural information gain to the potential interprocedural gain.

We further split the comparison into three different categories of summary functions

Input Summary Function of Flow Nodes: The input summary functions of flow graph nodes are important because they are the only intraprocedural summary functions which have to be shipped in the certificate. Each flow graph node contains a single sequence of instructions so that the intermediate summary functions can be reconstructed easily.

Output Summary of the Exit Node: The output summary of the exit node is the interprocedural summary function of the whole method. The open representation computed in the intraprocedural computation phase contains all relevant dependencies on callee summaries and forms the basic data structure for the interprocedural summary computation for the whole software module.

Input Summary Functions of Call Instructions: The input summary functions of call instructions are important because they are used during the final interprocedural computation phase. The goal of this phase is to compute a safe approximation of the invocation context of each method, and the input summary functions of call instructions directly map the invocation context of the caller to the invocation context of the callee at the specific call site.

We start now with the investigation of the input summary functions of flow graph nodes before we proceed to the other kinds of summary functions.

Input Summary Functions of Flow Nodes Input summary functions of control flow nodes are important to encode and ship the intraprocedural part of the result. They map the invocation context of the method directly to the input state of the flow graph node and comprise the effects of preceding branches and loops. In contrast, the summary functions of points within a flow graph node can be reconstructed comparatively easy, because this requires the subsequent consideration of the instruction-level summary functions of the straight-line code within the node.

In contrast to the input summary functions, the input states of the intraprocedural result do not have to be shipped in the certificate because they can be immediately constructed from the intraprocedural summary functions and the invocation context representation.

The program state which is mapped by the summary functions consists of the local variables and the operand stack in the current implementation. In order to decrease the size of the function representation the framework does not store expressions which model the identity mapping of a data flow variable. This is useful because the data flow information for many data flow variables remain unchanged for significant parts of the method. For example, parameters are usually not assigned new values.

In contrast, we expect that most of the data flow variables which represent the operand stack are mapped to the most pessimistic element because the operand stack of the Java virtual machine is usually empty when a branch instruction is executed. There are very few source-level language constructs which are compiled to a bytecode sequence which produces a non-empty stack. One example is the conditional assignment operator "?", which is not used excessively. Other examples include the results of boolean negation or comparison if they are used as method parameters or stored into fields. In fact we found in all pieces of subject software that over 99% of the operand stack

variables are mapped to the most pessimistic element. We conclude from this observation that the data flow through the operand stack affects the analysis only within the flow graph nodes. Thus, the summary function representation can be condensed even further, if the implementation allows for the specification of different default mappings for different kinds of data flow variables: the default choice for local variables should remain the identity mapping while the default choice for operands stack variables should be changed to the most pessimistic element.

The stack variables do not contribute a significant amount of information. Thus, we have to inspect the mappings of local variables, to estimate how much data flow information is derived from the intraprocedural context and what amount of information can still be gained by the subsequent interprocedural analysis phases. Figure 9.2 shows the percentage of the different kinds of defining expressions for each piece of the subject software.

The percentage of the most pessimistic expressions ranges from 47% to 73%. The high number of pessimistic expressions is not surprising, because the current implementation of the framework makes pessimistic assumptions for language constructs like the access to fields and the copy constant propagation additionally treats the result of arithmetic expressions pessimistically.

The distribution among the other kinds of expressions is more interesting. Most notable, the analysis uncovers many copy constants from the intraprocedural context already. The Java Card library exhibits the highest rate of copy constants in the local variables. More than 6% of the defining expressions in the function representation are copy constants [1]. This corresponds to the observation that the implementation of the Java Card platform sacrifices object-oriented design principles due to efficiency concerns and solves several problems by a direct manipulation of integer values to avoid the overhead of additional object instantiations. The same reason explains the difference between the other runtime libraries and the modules of our analysis framework. The current prototype implementation of the framework represents even central data structures like summary functions in an object-oriented style and does not operate on many integer values. This behaviour may change if technical optimisations are introduced to increase the runtime efficiency but renders the framework an uncomfortable target for the copy-constant propagation for the time being.

The high number of intraprocedural copy constants stems from a code generation pattern in the standard Java compiler. The introduction of an integer constant translates to a bytecode sequence where the constant is generated on the operand stack stored into a local variable and read from the local variable for further use. This is a straight forward technique but not necessary, if the constant value is used only once, e.g. to initialise a loop counter. The Java compiler does not strive for the optimisation of Java Bytecode, because a JIT compiler is expected to optimise the code on a standard target platform anyway. However,

[1] Remember that the evaluation does not count identity mappings in summary functions and that local variables which contain a copy constant throughout their lifetime appear in summary functions at several program points.

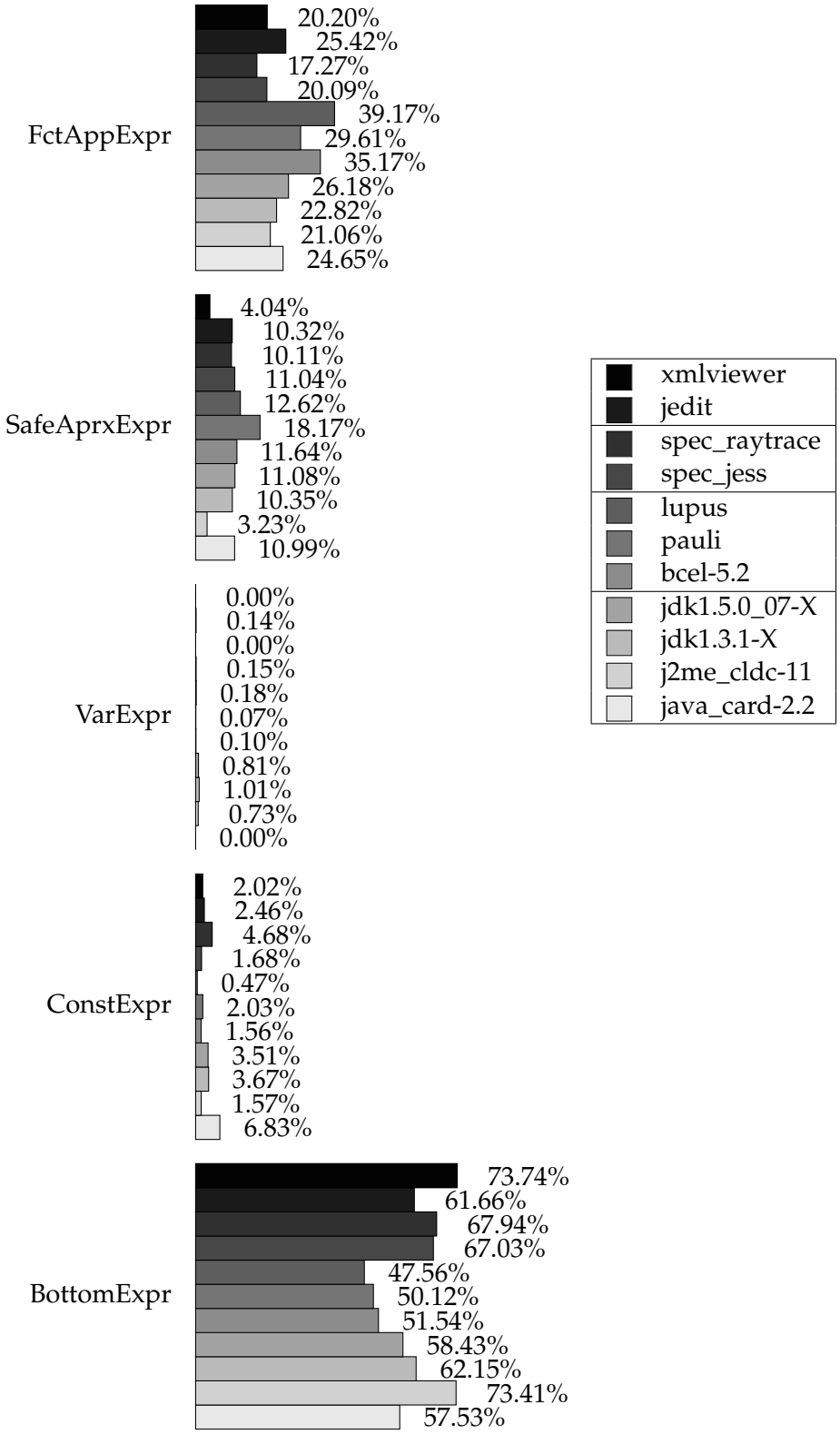


Figure 9.2: Local Variables in Input Summary Functions of Flow Graph Nodes

we cannot expect that a JIT compiler is part of a resource-constraint execution environment like a Java Card. Thus, the analysis results can be used to support some lightweight optimisations of the bytecode in such an environment at affordable costs.

Pessimistic and constant expressions together form the intraprocedural information gain, because they cannot change anymore during the subsequent phases of the interprocedural analysis. Though they dominate the other pieces of the result in the subject analysis, it is interesting to investigate the remaining data flow expressions. Firstly, most remaining local variables depend on a single function variable expression. Thus, they depend directly on a single method invocation. The number of such expressions is the converse of the situation for constant expressions: the parts of the analysis framework exhibit more function expressions while their number reduces for the runtime libraries. Once again, we expect that this is an immediate consequence of the object-oriented style of the framework implementation, which for example uses delegation quite often. Up to 40% of the local variables depend on a function variable in the implementation of the LUPUS framework. This observation is even more interesting, because it does not only apply to the copy-constant propagation example, but to all analyses which use copy semantics to express the effect of assignments between variables. Thus, we expect that at least the same amount of function variable expressions will be observed for other analyses.

The number of safe approximation expressions is about 10%. Only the PAULI framework shows significantly more and the application programs and the Java Card library show significantly less safe approximation expressions. This is a consequence of several fundamental characteristics of the summary function approach. A safe approximation expressions states that the state of the corresponding data flow variable depends on several program paths. The higher the number of such execution paths, the more likely it is, that one of them reduces the variable state to a safe-lower bound - and all other paths are dropped. Furthermore, the contribution of complex interprocedural program paths is "hidden" by function variables, because a function variable is a placeholder for the effects of a complete method invocation.

The comparatively small number of variable expressions calls for an explanation. Local variable registers in the virtual machine which hold parameter values do not change most of the time, which leads to a high number of identity mappings in the summary function representation. Such mappings show *not* up in the evaluation result for the variable expressions, because identity mappings are not stored in the data structure which represents the summary functions. Thus, the values show where a local variable holds the same value as *another* local variable. The investigated samples show, that such situations usually stem from non-empty operand stacks at branches - which exist but are very rare as we already observed earlier.

We conclude that the potential interprocedural information gain which can be achieved by considering the data flow via the result values of method calls ranges form 25% (Java Card library) to 50% (LUPUS framework) of the local

variables in the input summary functions of the program. We investigate the achieved information gain in Section 9.2.2.

Interprocedural Summaries The open representation of the interprocedural summary function of a method, shows how an invocation of the corresponding method changes the program state. We investigate the representation after the intraprocedural summary computation phase already, to find out, how many of these manipulations can already be inferred by the isolated analysis of single methods and how many effects depend on an investigation of the interprocedural data flow.

The manipulations of the program state by a method invocation have to be discriminated into two classes: the modifications which become visible in the caller and those which become not. For the time being, the return value of a method is the only variable which modifies the program state in the caller. Fields are not yet modelled, and the local variables and the operand stack of the callee are invalidated upon method return when the method frame is removed from the call stack. Therefore, the evaluation focuses on an investigation of the defining expressions of the method result variable in interprocedural summary functions.

Figure 9.3 shows the different kinds of data flow expressions which define the result value of a method after the intraprocedural part of the function analysis. More than halve of the methods are known not to return a constant value after the intraprocedural summary function analysis already. Some application programs even exhibit more than 80% of pessimistic expressions. This situation is not surprising because the pessimistic treatment of field accesses which specify for example the result of wide-spread getter-methods, are treated pessimistically. Furthermore, the method result is the final state of the method invocation and is more likely to depend on some non-constant value, than a local variable which is initialised and used within the method only.

The fact that the analysis uncovers copy-constants during the first phase of the analysis already is astonishing at the first glance, because this implies that the corresponding method returns a constant value. An investigation of the situation revealed that it stems from two programming patterns: Firstly, some methods return a constant value to indicate the normal termination of the method while the erroneous termination is indicated with an exception. The Java Card library uses this pattern extensively and contains 449 Java methods only, which leads to the high percentage of constant result expressions. Secondly, some methods override abstract methods in a special way. For example, the method which yields the number of subexpressions in an atomic data flow expression in the LUPUS framework returns the constant one.

Similar to the situation for input summary functions, the variable part of the result depends mostly on function variable expressions. Thus, it is likely that the result of a method invocation directly depends on the result of another method invocation. This can happen for example if one method just delegates

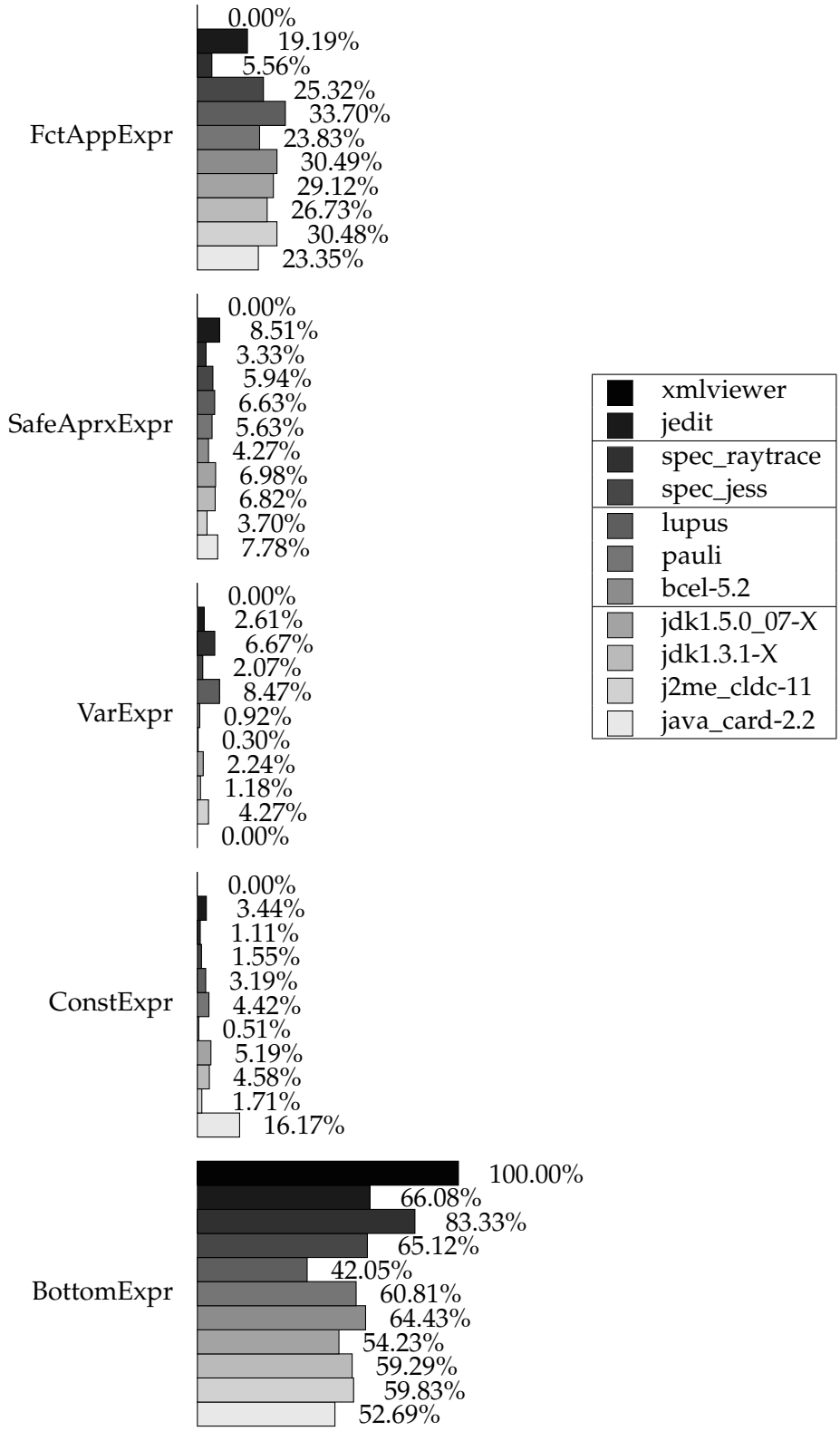


Figure 9.3: Result Values after Intraprocedural Function Analysis

the call to another object which happens quite often in object oriented programs. Once again more complex safe approximation expressions increase the variable part of the result by another 10%, so that the overall percentage of the variable part of the result ranges from 24% (xmlviewer) to over 50% (LUPUS-framework). Once again the results hint at the fact, that the extensive use of object-oriented design principles in the generic frameworks lead to a greater potential effect of the interprocedural part of the analysis. However, even the Java Card library and the J2ME library which target resource-constraint environments still have a potential interprocedural information gain of about 25%.

The variable expressions have to be interpreted differently for result functions. If the result of a method invocation depends on a single variable only, then the method returns one of its parameters. Such a situation stems mostly from a programming pattern where the method returns its own receiver reference for convenience. For example, the `StringBuffer.append()`-method returns the reference to the buffer which is the receiver of the call[2].

All in all, the results show, that a significant amount of interprocedural summary functions depend on interprocedural data flow, even if a simple analysis like copy-constant propagation is considered. This is an important observation, because it applies to all other analysis which use copy semantics for variable assignments.

We investigate the concrete information gain which is effectively achieved by the subsequent interprocedural analysis in Section 9.2.2.

Input Summary Functions of Call Instructions Input summary functions of call instructions directly map the invocation context of the caller to the program state immediately before the execution of the call. Especially, they contain defining expressions which specify the arguments of the call. The data flow information about arguments is interesting for two reasons. Firstly, the parameter expressions of the call are substituted into the summary function of the caller during function composition. Thus, more precise parameter expression can yield more precise output summary functions for the call site during the functional part of the interprocedural analysis. Secondly, the input summary functions of call instructions form the transfer functions of the final value computation phase during which the invocation context of a caller has to be mapped to the program state at a call site. This state contributes to the safe approximation of the invocation context of all call targets. Thus, the input summary functions of call instruction provide a first intuition about the potential precision of the invocation contexts.

The data flow information about the arguments of a call is contained in the mapping of the operand stack variables, because the Java virtual machine moves all arguments onto the stack immediately before the call. The measurements

[2]The receiver reference of the code is stored in a local variable register on the bytecode level like parameters and other local variables at the source-code level. As a consequence, the bytecode analysis treats all different kinds of source-level variables uniformly.

in Figure 9.4 count the defining expressions for all operand stack elements. This is a superset of the arguments because the operand stack can contain additional elements at a call site. However, this is not likely to be the case, because the bytecode produced by the standard Java compiler uses the operand stack usually only to supply the operands for the next instruction. Again the measurements show that most of the data flow information about arguments is retrieved in the intraprocedural phase already: between 73% ad 80% of the elements on the stack correspond to the most pessimistic element.

Remarkably, the percentage of variable expressions is quite large in comparison to the other kinds of investigated summaries. It ranges from 7% to 16%. These numbers include the common situation that the receiver reference of the caller is used as the receiver reference of the callee.

At least some of the argument values are copy constants, which contribute 2% to 7%. Again, we observed the highest rate of copy-constants for the JavaCard library which is due to the fact that the code of the JavaCard platform solves many problems on the level of byte- and short-values which would be solved in an object oriented style in a standard Java environment.

Summary The results after the intraprocedural analysis phase show that interprocedural analysis is promising, even if the analysis is as simple as a copy-constant propagation. Even though there are differences depending on which piece of the result is considered, usually more than 20% of the whole result depends on interprocedural data flow. The part of the result which is not fixed after in the intraprocedural context can even reach up to 50% as we have observed for the input summary functions of control flow nodes in the result of the LUPUS framework. Thus, there is a significant *potential* information gain which can be achieved by an interprocedural analysis.

The amount of data flow information which depends on interprocedural data flow will increase further, if the default modules of the framework start to consider for example the data flow via fields. Furthermore, it is possible to increase the expressiveness of a concrete analysis by additional problem-specific improvements. For example, a linear constant propagation can increase the number of integer constants because it symbolically evaluates arithmetic expressions which take a constant operand.

We do not continue the evaluation along these lines but continue to investigate the different analysis phases of the copy-constant propagation. The goal is to determine how much of the potential information gain detected after the intraprocedural summary phase is already achieved by the subsequent interprocedural analysis.

9.2.2 Interprocedural Summary Computation

The interprocedural summary computation resolves the calling relations within the software module. To do so, the function variable expressions which have

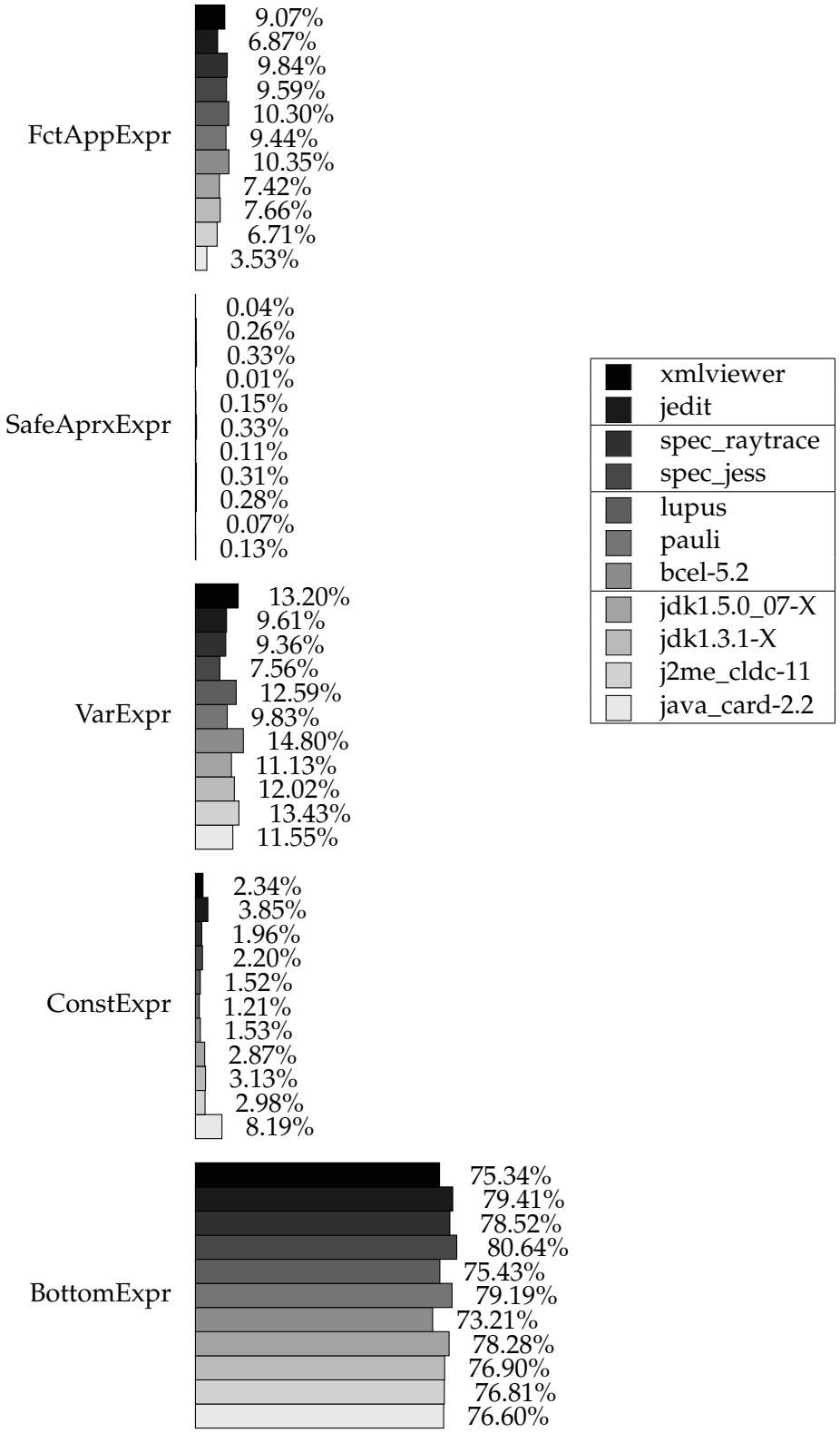


Figure 9.4: Operand Stack Expressions of Input Summary Functions of Call Instructions

been computed in the first intraprocedural analysis phase are substituted by the summary functions of internal callees. Thus, the function variable expressions of the intraprocedural result can change in three different ways.

Firstly, they can reduce to the most pessimistic expression, or to a constant value. In this case, the result of the call is fixed already even though the subsequent interprocedural invocation context computation has not yet been performed. In other words, the result of the method invocation does not depend on the invocation context of the call. This shows that some pieces of the data flow result are already determined during the interprocedural analysis phase which is why we call such pieces of the result the *interprocedural information gain*.

Secondly, function variable expressions can reduce to expressions which contain data flow variables. Data flow variables refer to the invocation context of the caller. Thus, these pieces of the analysis result depend on the subsequent value computation phase, which computes safe approximations for the invocation context of each method.

Finally, a function variable expression can reduce to a function variable expression which refers to an external call target. This happens if a method call either directly or transitively depends on a call to an external method. A method call can target an external method if some of the potential receiver classes inherit a method implementation from an external class in the superclass chain or if some of the receiver types refer to a cone type which can be extended by additional classes. The first case has to be treated pessimistically always. The question whether or not a cone type can be extended by external code depends on the assumptions about the analysis scenario.

Now, we investigate the interprocedural information gain and the potential effects of the "worst-case"- and the "closed-program"-assumption, before we proceed with an investigation of the remaining influence of the final invocation context computation phase in the subsequent section.

Interprocedural Information Gain We measure the information gain from the interprocedural summary function computation by a comparison of the constant expressions and the most pessimistic expressions in the summary functions after the intraprocedural and after the interprocedural function computation phase.

The measurements in Figure 9.5 show the differences of the defining expressions of the summary functions after the intraprocedural and interprocedural analysis phase. We focus the evaluation on the input summary functions of flow graph nodes, because we observe the most significant changes for this kind of summary functions. The values show how the relative number of an expression type changes: for example, the 6.06% decrease for the function variable expressions is a decrease of the intraprocedural 20.20% (refer to Figure 9.2) to 14.14%.

Essentially, the numbers show that the interprocedural summary function computation phase reduces most of the safe approximation and function variable expressions to safe lower bounds. This is not surprising for the copy constant

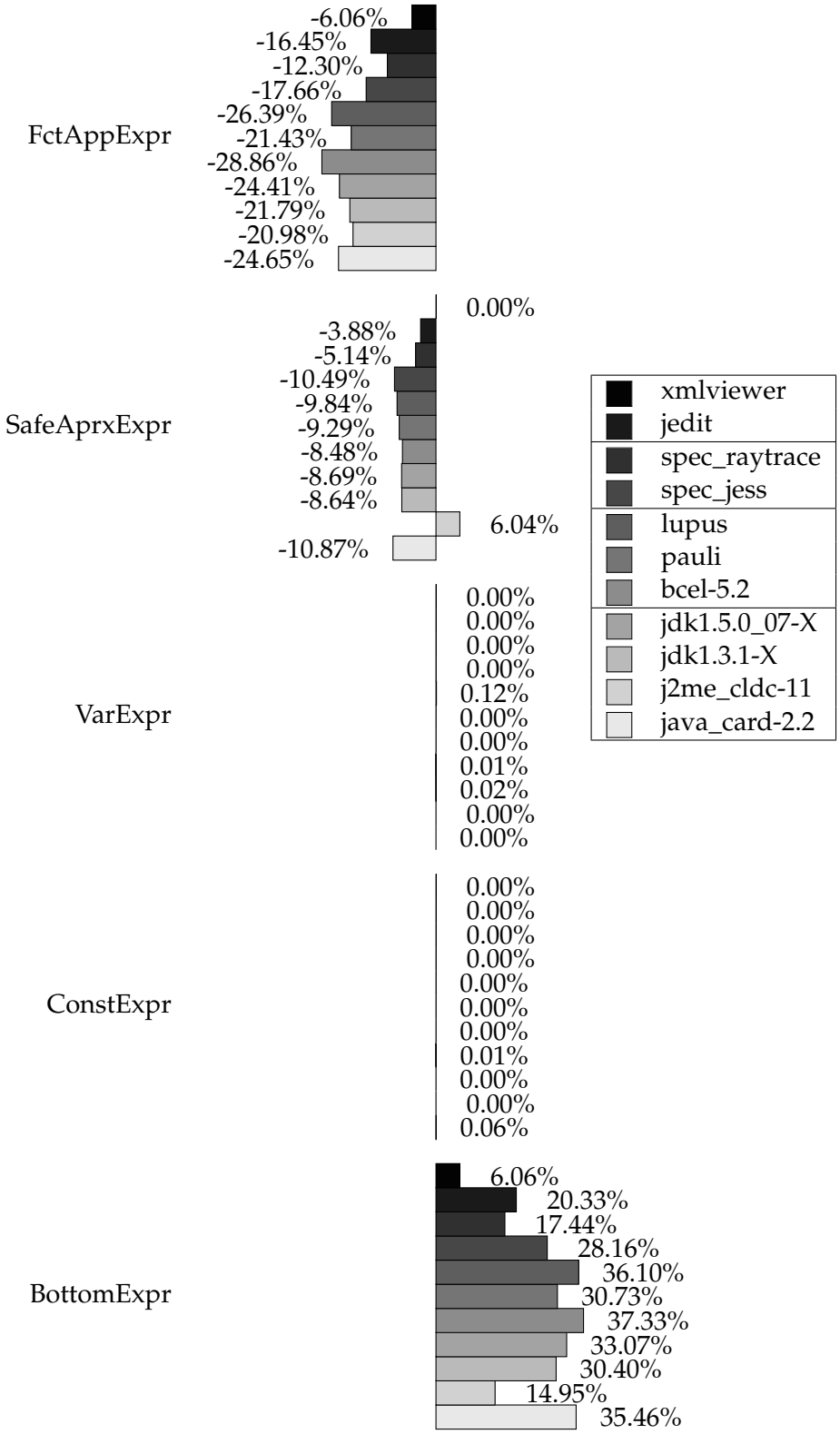


Figure 9.5: Changes in Flow Node Input Summaries after Interprocedural Analysis

propagation because if a data flow value depends on a method invocation it is not likely that the result of the method invocation will turn out to be a constant value. Interestingly, the analysis detects such unlikely situations in the JavaCard library because the percentage of constant expressions increases slightly by 0.06%. The explanation is that the programming pattern to return a constant integer to indicate the normal termination of a method affects the local variables in the caller.

There is also a small increase in the number of variable expressions which is a consequence of the insertion of summary functions which return their receiver reference into the summary function of callers.

Remarkable is the increase in the percentage of safe approximation expressions in the analysis result of the Java Micro Edition. The percentage of safe approximation expression increases if a single function variable expression is substituted by a defining expression within the callee summary that depends on different control flow paths in the callee. For example, the result of a method can either be a constant or the result of some external method invocation.

All in all, the interprocedural analysis shows that most data flow values which depend on method invocations reduce to the safe lower bound during the analysis of the software module. This is quite natural for the very conservative copy-constant propagation implementation in the LUPUS-framework. However, the interprocedural analysis is able to uncover interprocedural data flow even in such a unfavourable setting. Furthermore, the results show that an application scenario where the code producer attaches the analysis results of a sufficiently large software module is interesting because many internal data flow dependencies are treated by the modular analysis already. As a consequence, the number of complex function variable expressions reduce to those which depend on external methods only, and the rest of the defining expressions in summary functions becomes structurally simple. This reduces the size of the function representation in the certificate, while the approach preserves useful information and even dependencies to other modules.

Effects of Safe-Approximation Strategies The first and second analysis phase compute the influence of the calling relations within the software module. However, a small number of function variables remain which may depend on external method implementations. The analysis phase can deal with such dependencies in several ways.

Each call which targets an inherited method from an external superclass has to be treated pessimistically always. In contrast, a receiver type expression which leads to external calls because it contains cone types can be treated differently depending on the way the class hierarchy can be extended.

The worst-case assumption expects that the class hierarchy can be extended in an arbitrary way. Thus, any cone type can refer to some external subclass that contributes a new implementation. As a consequence all function variables have to be replaced by safe lower bounds.

In contrast the closed-program assumption expects that classes in the software module cannot be extended which is roughly equivalent to a situation where all classes in the software module are implicitly expected to be *final*.

Figure 9.6 shows the differences of the expression distribution in the final summary functions after the remaining function variables have been treated according to the "closed-program" and the "worst-case" assumption respectively. We observe that the removal of function variable expressions due to the application of the closed-program assumption uncovers variable expressions, constant expression as well as safe approximation expressions which then consist of several variable and constant expressions.

However, the percentages of the variable, constant, and safe approximation expressions differ by less than 1%. An exception is the result of the raytrace application in the spec benchmark suite where the differences exceed 2%.

Again, the results show the interprocedural approach can uncover valuable data flow information but its effectiveness is restricted due to the pessimistic nature of the copy-constant analysis. Furthermore, the comparatively limited number of data flow values which depend on external method invocations due to dynamic method binding justify the decision to stick with a simple mechanism for the resolution of dynamic call in the first prototype implementation of the framework.

9.2.3 Invocation Context Computation

Modular analysis is a challenging task, because the effects of external code can influence the achievable analysis precision significantly. The modular setting influences the way we have to deal with the resolution of dynamic calls, as discussed in the previous section. The closed-program assumption exploits that reasonable assumptions about the loading of additional classes can be made which rule out several call targets even if the type information about the receiver type not very precise.

It is difficult to apply the same principle for the invocation context computation as well. Essentially, we cannot easily restrict the potential entry points into the software module. A potential entry point is a method of software module which can be called by external code. A modular analysis cannot investigate the corresponding call site in the external code. Therefore, a modular analysis has to make worst-case assumptions about the invocation contexts of all entry points. If the modular analysis expects that all methods are potential entry points, then this pessimistic assumption renders the invocation context computation useless.

Thus, the situation calls for an adopted version of the closed-program assumption - i.e. a strategy which allows for a reasonable restriction of the entry points of the module. We envision strategies which exploit special language features. For example, private methods are visible in the scope of their defining class only. Therefore, only the methods of the defining class can contain call sites and it is not possible to extend the class with additional methods after the class is

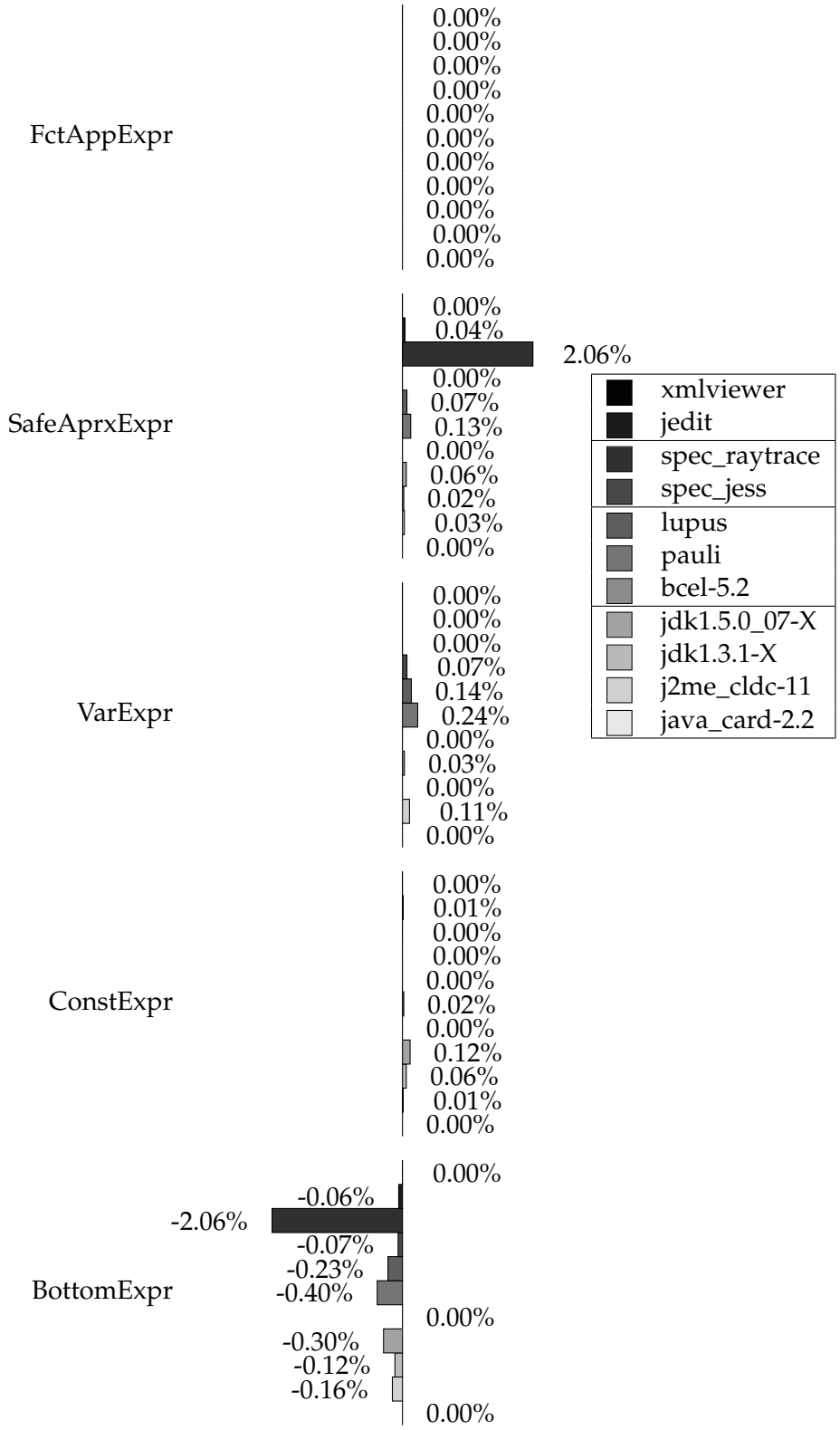


Figure 9.6: Differences between Worst-Case and Closed-Program Assumption for Local Variables in Input Summary Functions of Flow Nodes

loaded. This simple observation rules out private methods as potential entry points into the program module. However, the invocation contexts at the call site of a private method can depend transitively on the invocation context of some public method which contains the call, so that the potential information gain from the invocation context determination will presumably still be limited.

The current implementation of the framework does not support sophisticated strategies for the restriction of the entry points into the program. Thus, we have to restrict the evaluation to the investigation of the *potential information gain* of a analysis phase which aims at the computation of more precise invocation contexts. This can be achieved by an inspection of the final summary functions derived in the functional part of the analysis. These summary functions contain data flow variables wherever the result of the summary function depends on the invocation context of the method. Thus, we can determine how many pieces of whole analysis result depend on the invocation context computation.

Figure 9.7 shows distribution of expressions types in the final input summary functions of flow nodes after application of the closed-program assumption. The interprocedural function analysis and the subsequent treatment of external code has already fixed more than 90% of the defining data flow expressions. However, the evaluation result does not contain identity mappings which represent unmodified parameter values. Anyway, we conclude that the effect of more precise values for the invocation contexts is rather limited for the copy constant propagation problem.

To further support this claim we additionally inspect the invocation contexts at each call site in the software module. The subsequent value computation phase can compute an invocation context at a call site by the intraprocedural summary function which maps the invocation context of the caller to the specific point.

Figure 9.8 shows the distribution of expressions types in the final input summary function of call instructions after application of the closed program assumption. Again the interprocedural analysis reduces more than 80% of the defining expressions to safe-lower bounds. Only up to 15 % of the data flow values depend on the invocation context of the caller and usually less than 5% are copy constants.

If we take additionally into account, that there are usually several call sites of a single method and that the defining expressions for all of these call sites are merged by a safe approximation we can conclude that not many copy constants will survive the context computation phase, even if a reasonably precise strategy for the restriction of the entry points is available.

Again the result seems to be first and foremost a consequence of the pessimistic nature of the copy constant propagation. However, the general methodology which has been applied to the investigation of the copy constant propagation can be applied to other analysis as well. One of the most interesting parts of such an evaluation is the comparison of the influence of the modular setting and the analysis itself. We observe that the copy constant propagation is already inherently pessimistic so that pessimistic assumptions about external code do

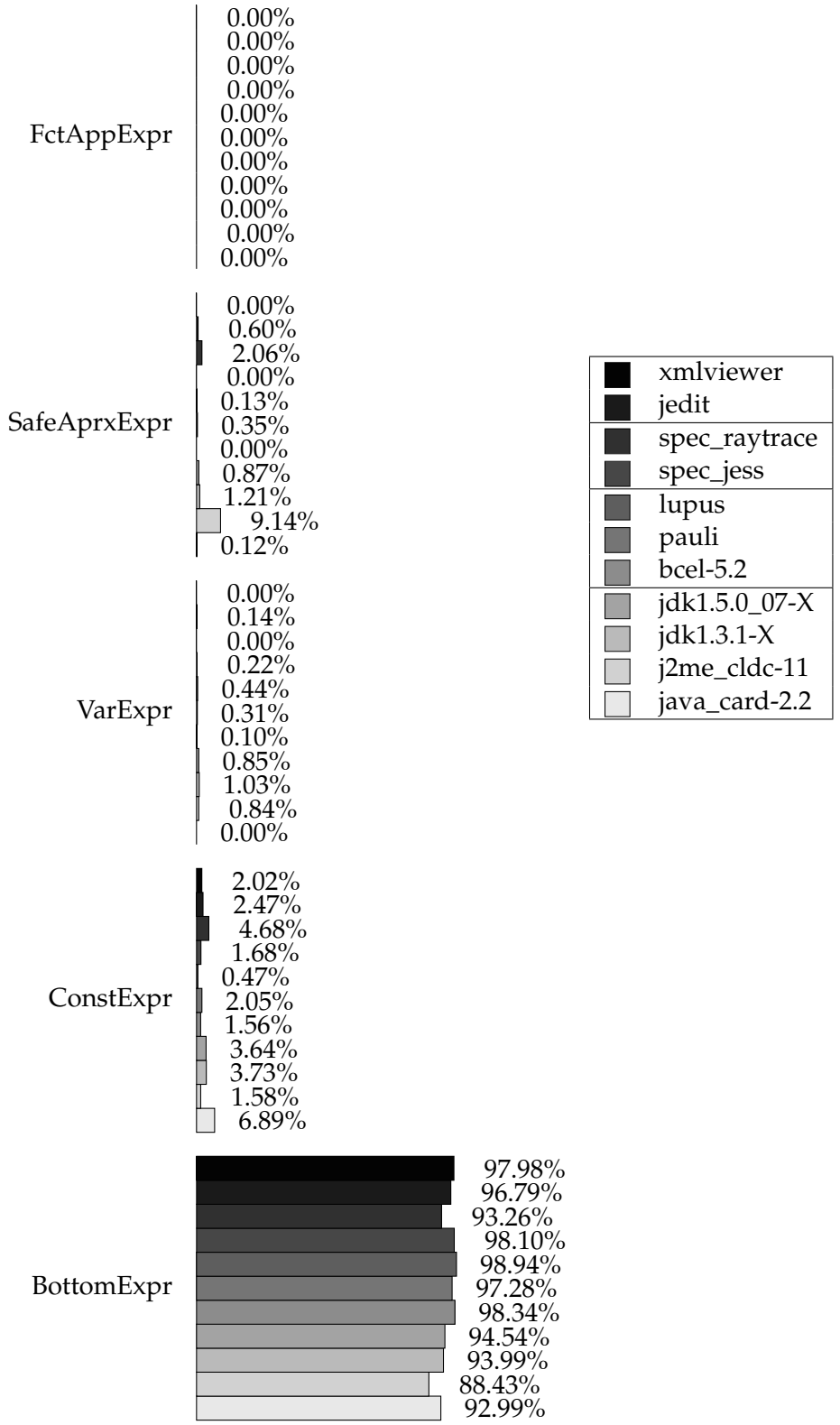


Figure 9.7: Final Input Summary Functions of Flow Graph Nodes (Closed Program Assumption)

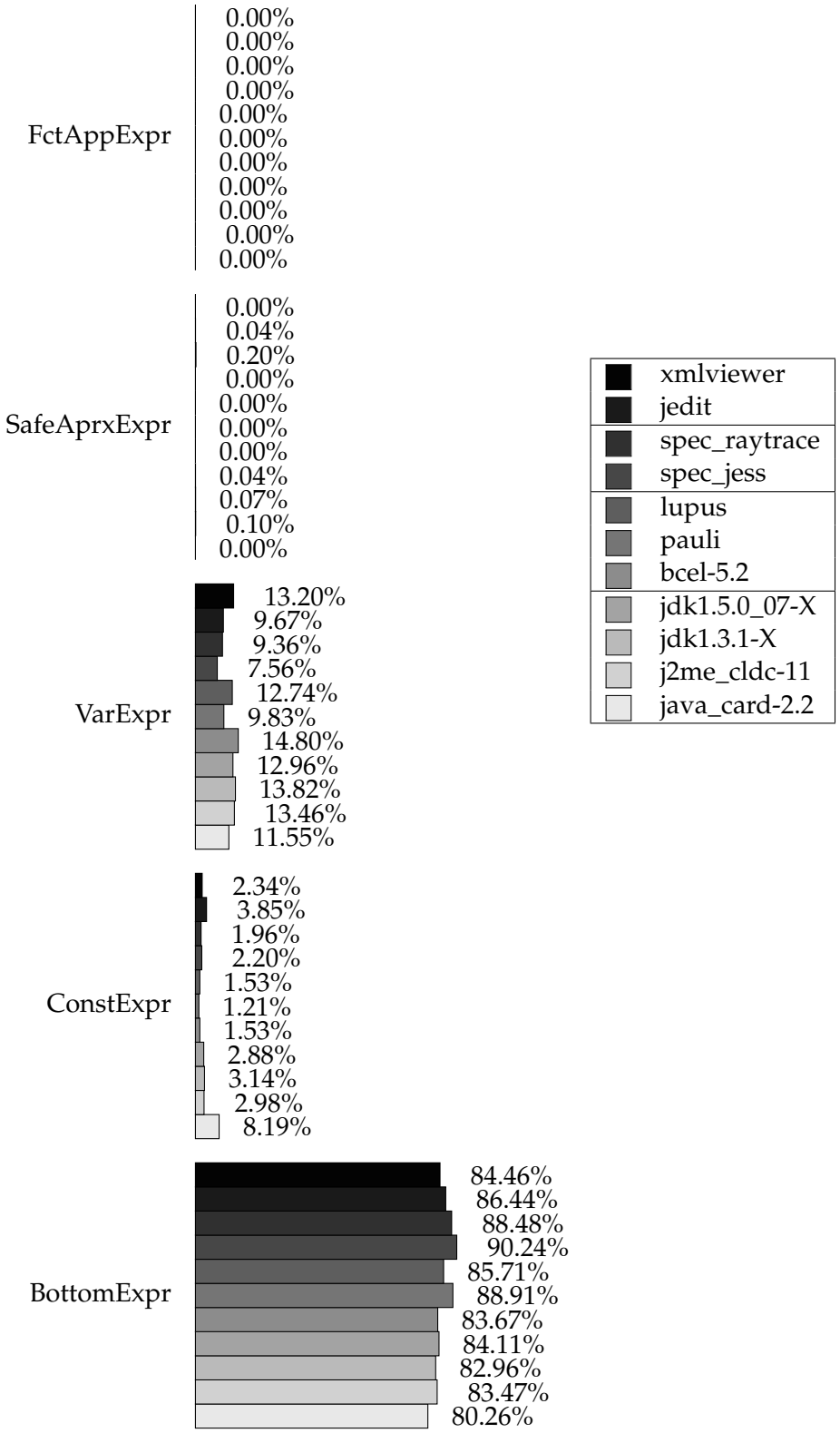


Figure 9.8: Final Input Summary Functions of Call Instructions (Closed Program Assumption)

not influence the result significantly. Other analysis may turn out to be more precise, so that more dependencies on external code remain. This in turn can increase the influence of the strategies which are applied to deal with the potential effects of external code.

9.3 Size of the Certificate

The certificate which is attached to the code enables the validator to reconstruct and to check the data flow result. The data flow result consists of one interprocedural summary function for each method, one intraprocedural summary function for each control flow node within the method, and a safe approximation for the invocation context of each method.

The size of the certificate depends primarily on the interprocedural summary functions and the invocation contexts because intraprocedural summary functions can be reconstructed comparatively easily. Most control-flow graphs exhibit an inherently linear structure so that the reconstruction of the input summaries can be achieved almost as easily as the reconstruction of intermediate summary functions for the straight-line code with flow graph nodes. The output summary of a predecessor node can act as an initial guess for the input node of a successor. This initial guess is likely to correspond to the final solution because most control flow nodes have only a limited number of additional predecessors which might influence the initial guess.

Furthermore, object-oriented design principles favour the decomposition of large method bodies into several smaller methods. In fact, we found that about half of the method implementations in the Java standard library consists of a single control-flow node only. Due to this general observations we focus our investigation on the size of interprocedural summary functions, before we investigate the size of invocation contexts.

9.3.1 Interprocedural Summary Functions

The size of a summary function representation can be inferred from the structure of the defining data flow expressions that encode the summary function. The representation contains a defining data flow expression for each single data flow variable. Thus, the size of the summary function is linear in the size of the program state representation even if all defining expressions are atomic.

The size of the *interprocedural* function representation can be reduced, because it is sufficient to store those parts of the summary functions which can influence the program state of the caller. This way, the current analysis setting allows for a reduction of the function representation to a single data flow expression because the result value of the call is the only piece of data flow information which influences the result. Therefore, we focus the measurements on the defining expressions of result values but remark that the size of the function

representation is likely to increase to the mapping of the whole program state if an analysis derives for example alias information or considers data flow via fields.

The size of defining expressions is not an issue in an application scenario, where the analysis at the producer side deals with all calls to external methods and removes all function variable expressions. Figure 9.9 shows the distribution of the expression types which define the result value of a call in the interprocedural summary functions after the interprocedural function analysis[3]. Most of the methods are either *void*-methods, or yield a pessimistic result. Constant and single variable expressions define most of the remaining summary functions so that a single data flow expression is sufficient to represent the summary functions. The small number of safe approximation expression can increase the memory requirements only if they combine a large number of data flow variables. However, we found that the number of data flow variables in safe approximation expressions is never larger than 3 for all pieces of the subject software.

Thus, less than a single data flow element is required to store the mapping of the result variable if the data structures do not store the safe lower bound explicitly. Therefore, we expect that the summary function representations stay linear in the size of the program state even if more data flow variables can be affected by a method invocation.

To summarise, the framework can deal with the simplest validation scenario where the final result of analysis which considers the software module in isolation is attached to the code, because we expect that a single function representation which is linear in the size of the program state suffices to encode the interprocedural summary function for each method.

In contrast to the simple validation scenario, the incremental validation scenario requires, that the validator additionally stores a valid *open* representation for each method until the validity of the final representation has been established. Open function representations are also required if modular results shall be combined at the consumer side in the partial validation scenario.

Therefore, we want to answer the question, if the certificate size is still manageable even if it contains the structurally more complex open summary functions which contain dependencies on callees in terms of function variable expressions.

The impact of function variable expressions on the size of the open interprocedural summary representations is most significant for the `jedit`-application. Figure 9.3 already provides a first hint because it shows, that the summary functions of the `jedit`-application contain the largest percentage of safe approximation expressions namely 8.51% (refer to Section 9.2). Safe approximation expressions can contain several function variable expressions if the result of the method depends on several method invocations on different paths. Furthermore, 19.19%

[3]The result values for `void`-methods are not shown but contribute to the overall method count. Therefore, the sum of the percentage numbers of a piece of subject software is less than 100%.

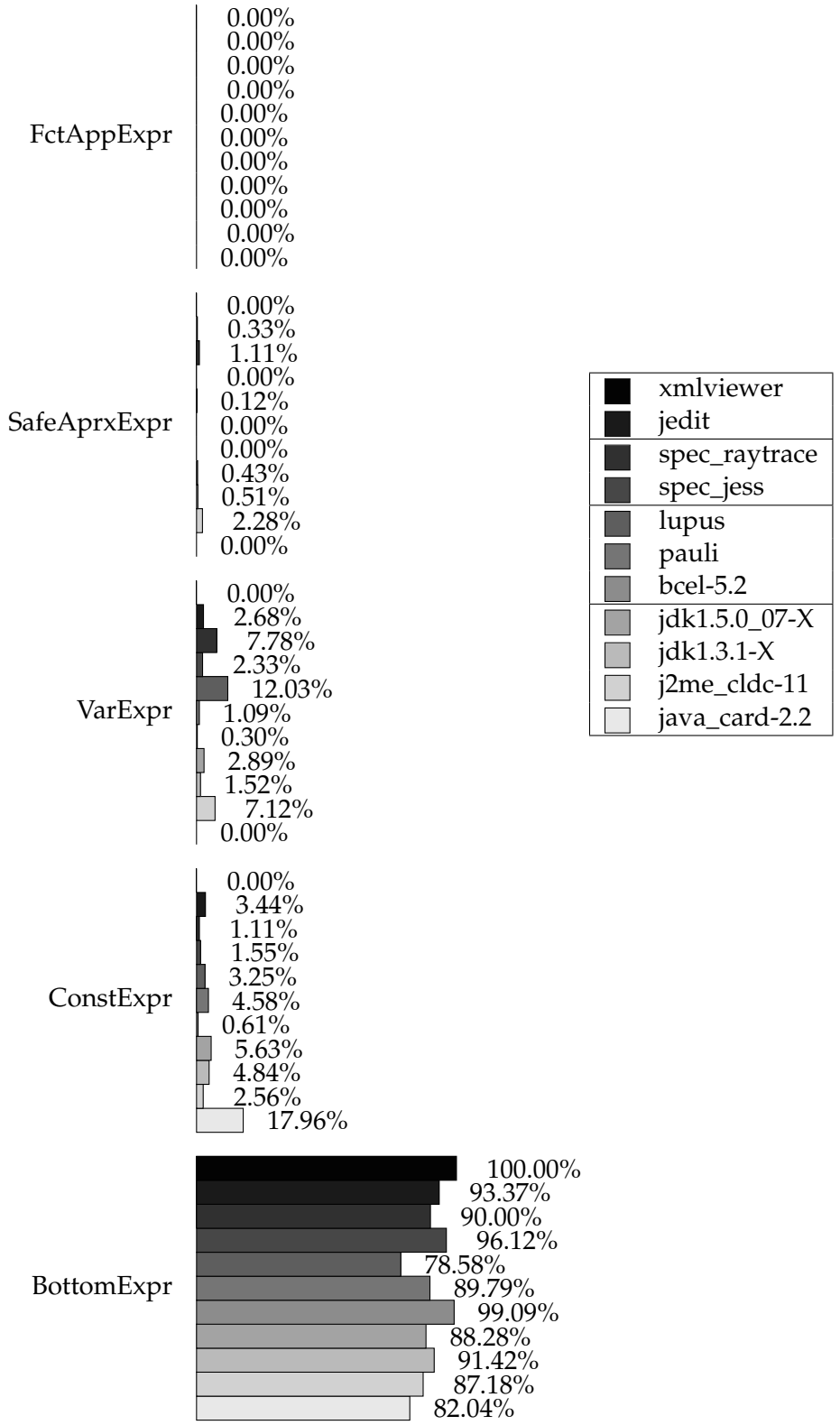


Figure 9.9: Result Values after Interprocedural Function Analysis (Closed Program Assumption)

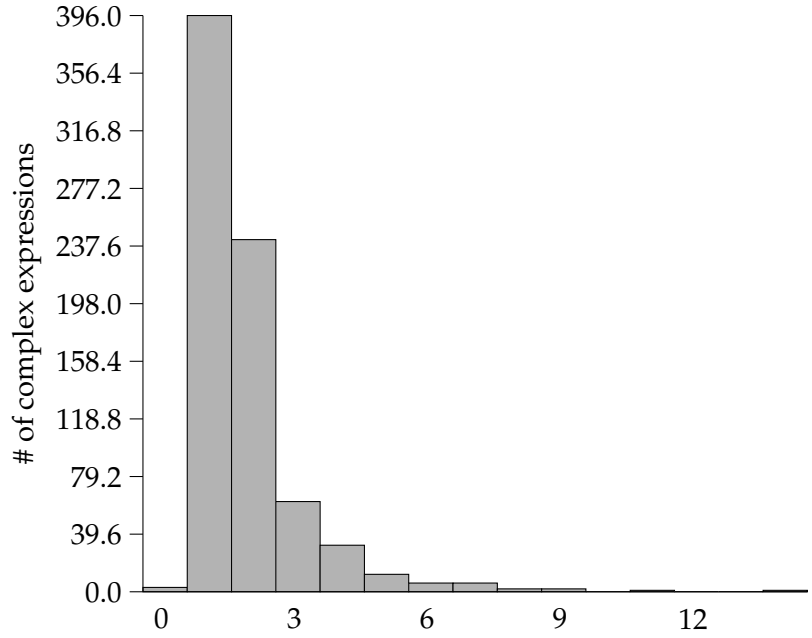


Figure 9.10: Number of Function Variables in Complex Expressions (JEdit)

of the summary functions depend on a function variable expression. A function variable expression can contribute more than one function variable, if its parameter state depends on some preceding calls.

In order to estimate the size of these complex expressions we consider the number of function variable expressions in complex expressions which is depicted in Figure 9.10. The average number of function variables in a complex expression is about 1.8, which can be observed for all other pieces of the subject software, too. Therefore, we expect that the size of the function representation will increase by a factor of three for complex expressions in open results, because the storage requirements of the parameter state of each function variable expression is roughly equivalent to an additional summary function (refer to Section 5.4.6). Thus, the certificate has to store three function representations for complex expressions on average and a single function representation for all other expression types. As complex expressions define about 20%-30% of the summary functions an average of 2 summary function representations per method should suffice to store an open function result.

This is less than the average number of invocation instructions per method, which is more than 5 for the `jedit`-application. The fact that function variables for some call instructions have been ruled out can have two reasons. Firstly, the corresponding function variable expression may have been dropped during function composition or during normalisation. Secondly, the design decision to limit the nesting depth of function variable expressions in the current implementation can rule out function variables. The first effect is a property of the

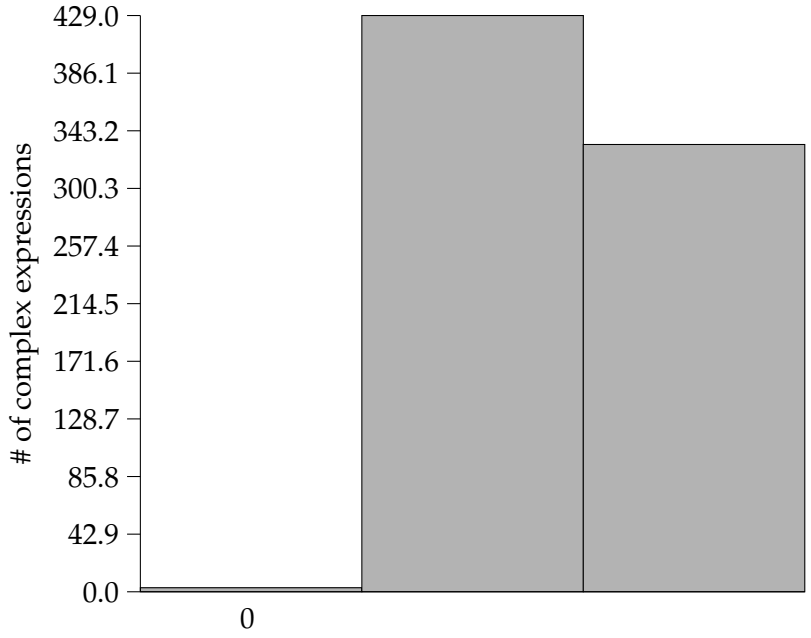


Figure 9.11: Expression Depth of Complex Expressions (JEdit)

constant-propagation analysis and any other analysis which deals with a significant number of safe lower bounds. In contrast, the second effect is a technical regulation that decreases the potential precision of the analysis. Figure 9.11 shows the depth of complex expressions in the open result after the intraprocedural summary function computation if the analysis framework restricts the nesting depth to 2. More than halve of the complex expressions have a nesting depth of one only. Therefore, we expect that the loss of precision due to the decision to restrict the nesting depth is limited - at least for analysis which show similar characteristics like the copy-constant propagation under consideration.

All in all, the certificate can store the final interprocedural summary result which results from a complete modular analysis at the producer site, easily because a single structurally simple summary function is sufficient for each method. The incremental and partial validation scenario requires larger certificates because it is necessary to store open summary function representations which encode a compressed variant of the flow graph of the method. The evaluation of the open summary functions for the copy-constant problem showed that the average number of summary functions per method increases to about 2. However, these summary functions are structurally more complex, so that we expect them to be linear in the size of the program state, while the final summary functions can be compressed to those pieces of the program state visible in the caller.

9.3.2 Size of the Program State

The size of the program state representation determines both the size of the invocation contexts and the size of the summary function representation which contains a mapping for each data flow variable in the program state.

However, two encoding strategies can reduce the size of the program state. Firstly, most pessimistic data flow values do not have to be stored in the representation explicitly. Secondly, the program state can be reduced to those parts of the program state, that can be affected by the method invocation. Conceptually, the program state contains the maximum number of local variables required by one of the methods in the analysis context. However, the relevant invocation context of a method just has to contain the local variables, that are used by the method in question.

We evaluate the number of local variables, to determine the consequences for the size of the program state representation. Figure 9.12 shows the absolute and average number of local variables in the methods of the whole lupus system including the Java 5 standard library. This is the largest piece of software considered by the evaluation and the results are quite representative for all other pieces of subject software as well. The average number of local variables is less

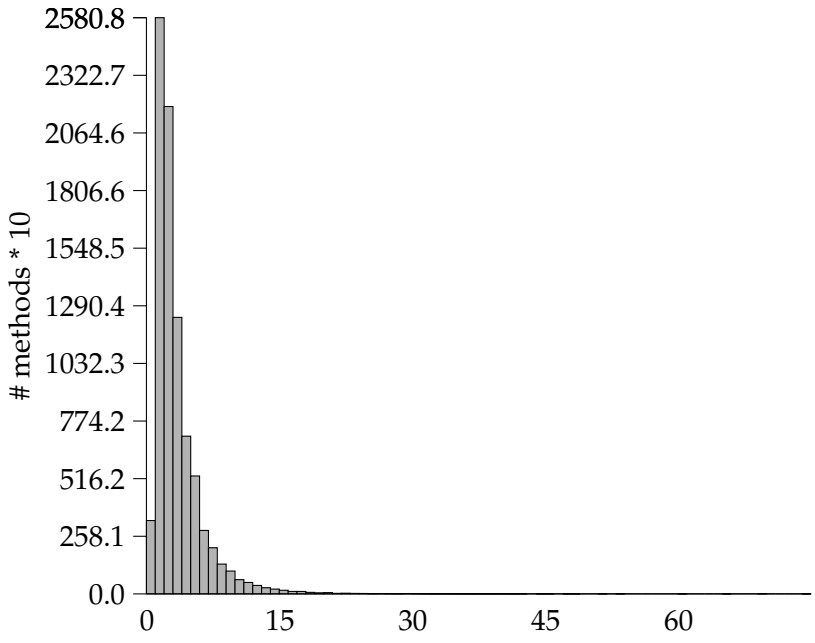


Figure 9.12: Local Variables per Method

than 3 but there are some extraordinary large methods, too. The largest method frame in the Java 5 standard library contains 74 local variables. The big number of local variables is not a challenge for the invocation context representation because even large methods usually take very few arguments.

However, some methods can be challenging with respect to the *intraprocedural validation* because a high number of local variables leads to large summary functions within the method. Furthermore, methods with many local variables tend to contain many flow graph nodes, so that the number of summary functions which have to be kept in memory during the intraprocedural validation can increase significantly.

Thus, we suggest that large methods are refactored into smaller ones which operate on their own set of local variables to decrease the storage requirements of the intraprocedural part of the validation. The fact that the interprocedural validation reintegrates the potential effects of the smaller callee into the summary function of the original function automatically is one of the advantages of an interprocedural analysis framework.

To summarise, we expect that the size of a binary certificate file is likely to contain very few information only, if the underlying data flow problem is as simple as the interprocedural copy-constant propagation. The central idea for an efficient representation is to restrict the information to the parts actually required for the validation process: The invocation context has to contain information about the parameters only, each interprocedural summary function can be restricted to the defining expression of the method result, and all expressions which correspond to safe lower bounds do not have to be stored explicitly either. Conceptually, all these technical improvements boil down to the application of the safe lower bound principle because the omission of irrelevant data flow information can be interpreted as implicit under-approximation of all pieces of the result which are not explicitly stored in the certificate.

9.4 Evaluation of the Validation Phase

Both the analysis and the validation use the same infrastructure which is currently implemented in a way which offers many opportunities for further optimisations. This situation provides advantages and challenges for a fair comparison of the analysis and the validation phase. On the one hand, the fact that both modules use the same infrastructure simplifies the comparison on the conceptual level because the design decisions made on the model layer affect the analysis and the validation in the same way. Furthermore, technical optimisations in the model layer immediately improve both phases. On the other hand, the proportional improvement of a more efficient implementation will likely be higher for the analysis phase than for the validation phase, because the analysis operates on more complex data structures and accesses them many times while a linear pass over the final - and structurally more simple - results suffices to perform the validation. Another source of potential improvements is the iterative fix-point solver which influences the analysis phase only. If its runtime efficiency increases, then this will decrease the gap between the analysis and the validation module to some extend.

Thus, we conclude that the current evaluation setting is in favour of the validation phase. We deal with this situation in the following way: Firstly, we

strive to obtain general statements about the differences of the analysis and the validation phase which do not depend on implementation details. Secondly, we try to measure the impact of implementation details as best as possible.

9.4.1 Memory Requirements

Interestingly, the discussion of the certificate sizes for the different validation scenarios paves the way for the comparison of the memory requirements of the analysis and the validation phase.

The most important observation is that the requirement to store an interprocedural summary function for each method dominates the memory requirements as soon as the subject software gets sufficiently large. Two observations justify this statement. Firstly, the memory which is used for the intraprocedural summary computation phase can be reclaimed as soon as the initial open representation of the interprocedural summary function of the method has been constructed or validated. Secondly, the number of methods in a software module usually outnumbers the maximum number of control flow nodes in a single method by far.

The analysis phase and the validation phase operate on different kinds of summary function representations in the simple validation scenario. The analysis phase uses the *open* function representation which encodes a compressed form of the interprocedural flow graph to derive a *final* function representation where all function variables are replaced by the summary function of internal callees or replaced by safe assumptions if they refer to external methods. The validation phase uses the final interprocedural function representation to construct the instruction-level summary functions of call instructions during the validation of a single method.

Thus, the difference of the memory requirements between the analysis and the validation phase conceptually stem from the different sizes of the open and final representations of interprocedural summary functions, which we already discussed in detail in Section 9.3. The result reveals that the open summary representation is significantly more complex than the final one, because function variable expressions require to store the parameter state of the call in terms of additional summary functions. We found that on average two summary function representations which are linear in the size of the program state are required to store the open summary function result of each method. In contrast, the final result requires to store a single summary function only,

Given that the average number of data flow variables which are necessary to define the program state within the method is less than four (see Section 9.3.2), we expect that on average 8 data flow expressions are sufficient to store the open result of a method while a single data flow expression is sufficient to store the final result in its compressed form.

The memory requirements of the validation phase can be reduced even further if the validator drops all interprocedural summary functions immediately after

the last call instruction of the corresponding method is processed. The effect of this optimisation has not be measured yet, because both the analysis and the validation phase use an arbitrary processing order in the current prototype implementation.

The above considerations hold for the simple validation scenario where the code producer processes all calling relations and only ships the final result of this analysis to the code consumer. The incremental and the partial validation scenario require the transmission of the open representation to the validator, too. In this situation the difference in the memory requirements no longer depends on the different representations but on the way the validator uses the open representation. The validator needs the open representation to defer the integration of callee summaries until their validity has been established. This way, the open representation stays valid and can be used to derive a safe lower bound for the final representation at any point in time. However, the validator receives the final result of the analysis phase, too. As soon as the safe lower bound of the valid open representation matches the result of the analysis, the final function representation is valid and the validator can drop the open representation. The current implementation does not feature an incremental validator yet, but the high number of pessimistic summary functions in the final result of the copy constant propagation justifies the assumption, that very few open representations will be required during the incremental validation process.

To summarise, the validation phase offers many opportunities to reduce the memory requirements of the preceding analysis phase. The central reason is that the analysis phase operates on summary function representations which encode many *potential* interprocedural data flow dependencies. During the analysis phase many of these dependencies are ruled out, so that the final result is structurally much simpler than the initial solution of the analysis phase. Obviously, this behaviour essentially depends on the existence of reasonable safe lower bounds, which safely approximate the potential effects of method invocations or on a limited lifetime of data flow facts. In contrast, the size of the summary function representation will increase if potential dependencies between many data flow facts are highly likely and where it is not easy to rule out or restrict the potential effects of method invocations. However, such kinds of analyses are not well suited for any modular analysis scenario where the analysis cannot investigate external code anyway.

9.4.2 Runtime Requirements

The comparison of the runtime efficiency of the analysis and the validation phase is difficult, because both phases currently depend on the heavy-weight infrastructure of the LUPUS framework prototype. This infrastructure models the summary function concept as directly as possible. Furthermore, the implementation is designed for simple expandability so that additional layers of abstractions impact the runtime efficiency. For example, the comparison of an

environment involves a look-up of the data flow lattice for each variable in the environment because the environment implementation is able to store values of different data flow problems at the same time. This mechanism is useful to inject a simple analysis of the size of the operand stack into an arbitrary client analysis in a way which is transparent to the user and the rest of the framework. However, a mature framework should provide more efficient data structures, whenever the overall impact on the runtime efficiency becomes significant.

Further improvements of the infrastructure will increase both the efficiency of the analysis and the efficiency of the validation phase. Nevertheless, the improvements will impact the analysis phase more than the validation phase, because the validation accesses the elementary data structures much less frequently than the analysis phase. This is an immediate consequence of the general observation that the validation performs a single linear pass over the final result of the analysis only and that the final result is structurally more simple than the open representation the analysis phase operates on.

In order to compare the analysis phase to the validation phase we measure the runtime of an analysis phase that computes an open interprocedural representation of the summary function of each method, applies different strategies for the treatment of remaining function variable expressions and substitutes this interprocedural summary functions back into the initial open result of the intraprocedural summary representation. The runtime of these phases are compared to the runtime of a validation pass that takes a full certificate of the result and validates it with the same strategy for the treatment of external calls.

The measurements have been performed on an 2.60 GHz Intel Xeon processor with 6 MB cache and enough main memory to store all intraprocedural summary functions to avoid their reconstruction.

Figure 9.13 shows the runtime results for the Java 5 runtime environment which is the largest piece of software considered in the evaluation. The results show several characteristics which can be observed for all other pieces of subject software as well:

- The construction of the initial open function representation and the computation of the interprocedural summary functions ("Interprocedural Analysis") dominates all other phases.
- The impact of the safe approximation strategy which is applied to the open representation after the analysis phase and during the validation is not very significant.
- The validation phase is up to 20 times faster than the analysis phase even if we do not take the back substitution time into account.
- The runtime requirements of the analysis phase are way beyond the runtime requirements we would expect for a simple analysis like copy constant propagation even if it is applied on a large piece of software.

The inconvenient runtime of the analysis phase requires a discerning investigation and explanation. Firstly, the prototype implementation always performs

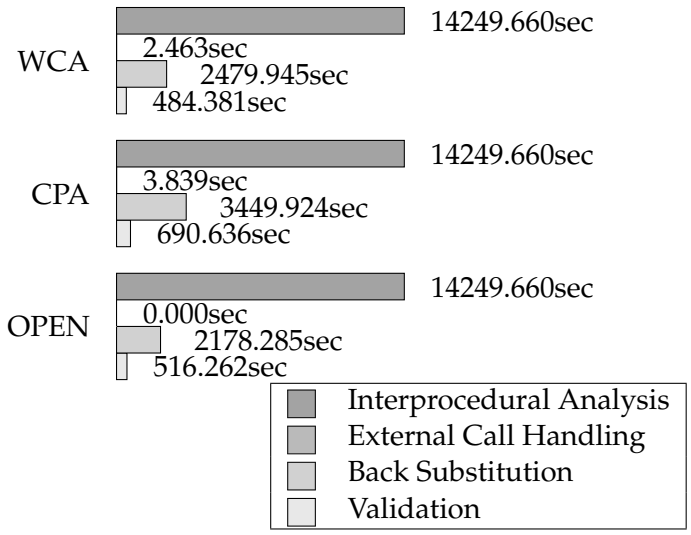


Figure 9.13: Runtime Measurements of the Java 5 Runtime Library

a modular analysis even if it intends to construct a final analysis result for the software module in isolation. The simple analysis scenarios which apply the "worst-case"- (WCA) or "closed-program"-assumption could do so immediately *during* the analysis phase which simplifies the function representations and speed up the analysis. Even though we have not measured this aspect we can already make an interesting observation about the validation phase which applies the safe approximation directly: its runtime requirements do not significantly depend on the way it has to deal with external calls. This is not surprising because a decision whether a call can have external call targets only affects the construction of the instruction-level transfer functions once for each instruction.

Another influence on the efficiency of the analysis phase is the interprocedural fix-point solver. For example, the current implementation of the fix-point solver processes the methods of a program in an arbitrary order and does not take the complexity of the summary functions into account. This leads to intermediate representations which are more complex than necessary. For example, it is advantageous to prefer an integration of callee summaries which do not contain any reference to function variables like leaf methods to simplify the representation of a caller by the normalisation of summary functions as early as possible.

Towards a Fair Comparison of the Analysis and the Validation Phase Even though this thesis focuses on the validation phase the current situation calls for a more in depth investigation of the runtime inefficiency of the analysis phase, because a runtime of more than three hours for a comparatively simple copy constant propagation hints at some conceptual problem in the implementation of the analysis.

In fact, a detailed profiling of the framework revealed that the combination of two factors give rise to the high runtime costs of the analysis phase: the heavy-weight implementation of the elementary data structures which represent data flow expressions and the conceptual decision to model the parameter state of a function variable expression explicitly in terms of an environment. In order to increase the robustness of the expression implementation, the prototype implementation creates copies of complex expressions whenever they are propagated during the analysis. This avoids potential side effects of subsequent manipulations but becomes a major issue, if the analysis phase explicitly constructs the parameter state of a function variable expression early in the analysis phase.

To explain the problem consider the example in Figure 9.14 which shows a method invocation m_1 that affects two different pieces of the program state at a join point. The current implementation of the intraprocedural propagation

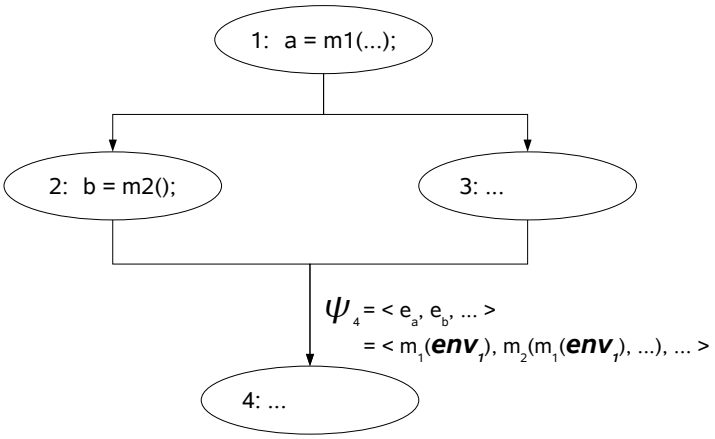


Figure 9.14: Duplication of Complex Environments by the Propagation of Function Variable Expressions

mechanism explicitly constructs the environment env_1 during the composition of the instruction-level summary function of the instruction in position 1. Furthermore, the expression $m_1(env_1)$ is propagated to point 2 where it contributes to the construction of the parameter environment for the function variable expression m_2. The construction of the parameter environment copies the defining expressions, in order to avoid that subsequent normalisations produce dangerous side effects on the original expressions. This increases the robustness of the system but yields the problem that changes in env_1 have to be propagated to all copies of the parameter environment.

Profiling of the current prototype implementation revealed that the construction and the update of nested function variable expressions contribute significantly to the runtime of the analysis phase.

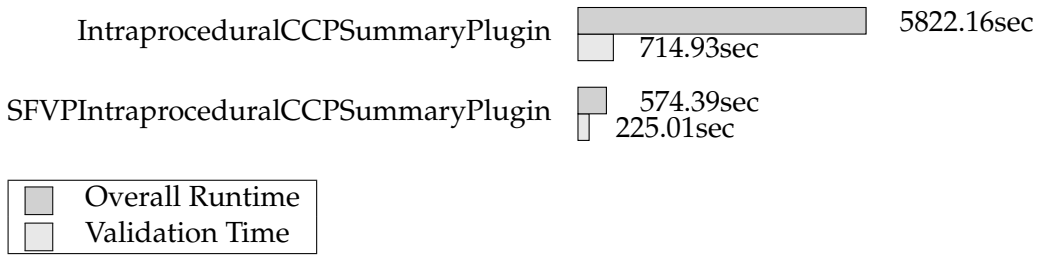


Figure 9.15: Runtime Comparison of Different Intraprocedural Analysis Phase (Java 5 runtime library)

A conceptual modification of the function variable representation can tackle this problem. Instead of explicitly constructing the parameter environment env_1 we can refer to the corresponding input summary function by a function variable ψ_1. Thus, the expression $m_1(env_1)$ is conceptually replaced by $m_1 \circ \psi_1$, where ψ_1 refers to the intraprocedural input function of the call instruction. As a consequence, the propagation mechanism does not produce copies of the environment env_1 but copies the function variable ψ_1 only. In contrast to the environment which can be a complex data structure, function variables which refer to input summary functions can be copied and constructed in constant time. Furthermore, changes to the input summary function ψ_1 do not have to be propagated to all dependent function variable expressions anymore - it is sufficient to update the function representation the function variables refer to. The new model also increases the efficiency of the interprocedural analysis phase because the substitution of callee summaries for function variable expressions can be deferred until they are either applicable or belong to a cyclic dependency which has to be resolved by a fix-point iteration.

This new idea to deal with the parameter environment of function variables is not fully integrated in the current prototype implementation of the LUPUS framework, yet. However, an implementation of the *intraprocedural analysis phase* which constructs function variable expressions according to the new model is available, so that we can compare the two implementations to estimate the potential effects on the runtime of the analysis phase.

Figure 9.15 shows a runtime comparison of the old and new style implementation (SFVP) of the intraprocedural analysis and validation phase. The new implementation is approximately 10 times faster. However, most of the runtime improvement is achieved in the analysis phase because the validation that uses the new model is only 3 times faster than the one which uses the complex function model. As a consequence, the runtime improvement of the validation phase drops to a factor between 1 and 2[4]. The runtime improvement of the validation phase is - though reduced - much more understandable: the conceptual improvement of the validation stems from avoided fix-point iterations

[4]In the current evaluation mechanism the analysis phase reuses the control flow graphs which have been constructed in the validation, so that the runtime advantage of the validation is slightly better than the given results suggest

and a copy constant propagation reaches its fix-points fast. Thus, a runtime improvement by a factor between 1 and 2 is more reasonable than a runtime improvement by a factor of 20 which we observed for the current prototype implementation.

We expect that significant runtime improvements can be achieved in the interprocedural analysis phase as well, because the manipulation of the parameter environments is a major issue in this phase as well.

Interestingly, the use of function variables which refer to intraprocedural summaries instead of an explicit construction of parameter environments offers another advantage: function variable expressions do no longer lead to nested expressions, which has been identified to be a major problem in Section 5.4.6. Therefore, a reformulation of the function expression model seems to be one of the most promising improvements of the functional approach to modular analysis presented in this thesis.

9.5 Summary

The results of our evaluation are twofold. Firstly, we establish an evaluation methodology that investigates the impact of the different sub-phases of the functional approach to interprocedural analysis and validation. This methodology uses the flexibility of the open summary function model, which expresses references to external code in terms of function variables and can be applied to all other analysis which are formulated in terms of the function model developed in this thesis. Furthermore, some parts of the methodology like the comparison of the intraprocedural and the interprocedural information gain may even be applied to other analyses which use the functional approach to interprocedural analysis. Secondly, the evaluation provides evidence that the validation of analysis results is useful in a modular analysis scenario, because the effectiveness of a modular analysis and the potential runtime and memory improvements of the validation phase both depend on the existence of reasonable safe lower bounds which restrict the potential interdependencies of data flow values.

The most important observations with respect to the evaluation methodology and the concrete results for a specific analysis can be summarised as follows. The inspection and comparison of the different kinds of open and applicable summary functions allows for the determination of several structural properties of a specific analysis:

- The *intraprocedural information gain* can be determined by a comparison of the constant and the variable part of open intraprocedural summary functions because only the variable part can be influenced by the subsequent interprocedural analysis.
- The *interprocedural information gain of the summary function computation* phase can be determined by the comparison of the constant and the variable part of the interprocedural summary functions which result from the functional phase.

- The *potential information gain of the value computation phase* can be derived from the inspection of the data flow variables which are referenced by the interprocedural summary functions.
- The effects of the *worst-case assumption* can be estimated by the safe approximation of all external function variable expressions in the interprocedural summary functions.
- The effects of the *closed-program assumption*, which expects that program classes are not subclassed and program packages are not extended, can be derived from the open summary functions, too. The approximation strategy just drops all function variables which refer to internal methods under the assumption that no additional subclass for a program class will be loaded dynamically. All other method invocations are still safely approximated.

Most evaluations of interprocedural analysis approaches focus on some of these aspects only. Usually, the implementation of an interprocedural analysis depends on some specific assumptions about the properties of external code, and these assumptions are implicitly encoded in data structures that are specialised for the analysis in question. Therefore, it is difficult to determine to what extent the analysis result depends on properties of the analysis problem, on properties of the dynamic call resolution, or on assumptions about the modular setting. In contrast, the functional approach and the summary function model developed in this thesis makes most of these different aspects explicit in the function representation.

The concrete evaluation of the summary functions of the copy constant propagation reveals that on average two summary functions per method are sufficient to represent the intermediate result in the interprocedural analysis phase and that a compressed variant of the final summary function of each method is sufficient during the validation phase in the simple validation setting. Furthermore, even the incremental and partial validation scenarios are manageable as long as it is possible to establish the validity of the final summary function result early for a significant part of the result.

The runtime measurements reveal that the current prototype implementation of the analysis phase suffers from the unfavourable design decision to model the parameter state of a function variable expression explicitly in terms of an large environment which is duplicated during the propagation of data flow values. As a consequence, the analysis phase is far from being competitive which is inconvenient even though the validation phase already runs comparatively fast. An astonishingly simple modification which replaces the explicit construction of a parameter environment by a reference to the intraprocedural summary functions which defines the state can tackle the problem. First runtime comparisons of different implementations of the intraprocedural analysis phase show that the new model reduces the runtime costs of this analysis phase to 10%.

Both, the memory and the runtime requirements are likely to increase if we consider analyses which are more complex than the simple copy-constant

propagation. However, we still expect that the approach is applicable to other analyses which are suitable in a modular analysis setting. The reason is that both, the efficiency of the summary function model and the efficiency of a modular analysis require that the potential negative impact of external method invocations can be kept under control. Only if this assumption holds for a concrete analysis, then a modular analysis yields significantly precise results and the number of function variable expressions in the summary function representations remains manageable. The implementation and evaluation of other analyses like the type inference analysis defined in Section 7.3 are the next natural steps which should be taken to support this general claim.

10 Conclusion

This thesis applies the proof-carrying code principle to separate interprocedural analyses from the use of their results in a safe way. This enables the use of analysis results in an inherently insecure network environment which connects devices with differing computational capabilities. The key observation is that it is easier to ensure that a given data flow result solves the system of data flow equations which specifies the underlying data flow problem than to perform the fix-point iterations which compute the result.

The result of an interprocedural analysis can be expressed in terms of summary functions. The central challenge for the validation approach is to find a function representation which allows for an efficient comparison of summary functions. We achieved this comparability by the definition of a unique normal form so that the comparison of summary functions reduces to a simple comparison of the internal structure of the summary function representation.

Another challenge for the validation approach is that it cannot rely on the results of auxiliary analyses but it has to ensure the validity of the auxiliary analysis as well. Therefore, we had to find solutions for the safe resolution of dynamic method binding which is a prerequisite for the interprocedural analysis of object-oriented programs. Furthermore, the capability to download additional code to a target platform is a central characteristic of the application scenario of this thesis. To deal with this issue, the analysis has to be capable to deal with separated software modules because not all of the code is available. The contributions of this thesis can be summarised as follows.

10.1 Contributions

The summary function model developed in Chapter 5 comprises the central methodical contribution of the thesis. First of all, the model supports the validation of interprocedural analysis results because the summary functions can be compared to each other easily. Essentially, one function safely approximates another if it contains more subexpressions within its defining expressions. Intuitively, we exploit that the safe approximation operation of any inducing data flow problem can only produce weaker results if it is applied to more data flow values.

This is not a property of a specific problem but a property of any data flow lattice. Therefore, it is possible to reuse the same function representation for different analyses in a generic way. More complex dependencies between data flow values can be expressed in terms of elementary transfer functions.

The function model currently treats elementary transfer functions symbolically and does not take any other properties than the applicability and the monotony of the functions into account. However, the model restricts the nesting depth of function expressions to a fixed depth which can result in a loss of precision. To avoid this loss of precision, the elementary transfer function model offers the opportunity to integrate problem specific normalisation rules. Essentially, elementary transfer function expressions emulate the micro functions of the graph based approach of of Reps, Sagiv, and Horwitz in a more flexible way. For example, it is possible to use elementary transfer functions symbolically if they do not meet all of the requirements imposed in the graph model. However, the necessary limitation of the nesting depth of elementary transfer functions results in a loss of precision which is can be avoided for the more restricted class of functions.

The function model does not only deal with the specification of data flow problems, but defines also normalisation rules and the support for the representation of modular results. The normalisation rules correspond to a partial evaluation of the constant terms in the defining expressions. They are closely related to path compression techniques because they strive to compress the data flow on different paths between two program points to an immediate mapping of the start state to the result state. The normalisation of summary functions solves one of the central challenges of the validation process because it reduces summary functions to a unique normal form. This is vital to ensure that the validator can compare the summary functions which specify the requirements of the data flow problem to the summary functions which represent the analysis results.

Function variables model the influence of external program parts in a modular result representation. The advantage of this novel approach is that it integrates the dependencies on external code directly into the function model. This way it is possible to rule out irrelevant external dependencies. Furthermore, function variable expressions can be safely approximated at any point in time which yields a safe under-approximation of the final result which shows that the general validation principles are applicable in the functional setting.

We successfully used the function model to define two data flow problems. Firstly, a copy-constant propagation which tracks the data flow of integer constant and `null`-references shows how it is possible to analyse the data flow on the call stack of a program. The analysis of the call stack is a prerequisite for more sophisticated analyses which also take the data flow via the object heap into account. Secondly, we augment function variables with type information about the receiver type of the call, in order to approximate the potential targets of a dynamically bound method invocation. This is a prerequisite for any interprocedural analysis of object-oriented programs because the runtime type of the receiver reference defines the target of a call, which in turn specifies the interprocedural flow graph of the program.

We use a precise type model to restrict the potential call targets even if the runtime environment allows for the dynamic loading of additional classes. The resolution mechanism is decoupled from the computation of the type

information by the implementation. In this thesis we specify two different approaches for the type computation: an adopted version of a traditional CHA-based approach which uses the "closed-program assumption" to deal with the expandability of the class hierarchy and a specification of a type inference algorithm in terms of the function model developed in this thesis. The first approach is sufficient for a first application of the framework while the second one shows how the analysis system can compute and utilise more precise type information.

All in all, the summary function model developed in this thesis solves the central challenges for the validation of interprocedural analyses results for software modules in an expandable object-oriented environment. The evaluation of the prototype implementation shows that the model is suitable to specify data flow analyses. The framework considers the data flow on the call stack of the program and implicitly constructs an validatable interprocedural flow graph for the subject software. Such a flow graph is a prerequisite for all more sophisticated interprocedural analyses which may follow.

The main contribution of this thesis is a methodical treatment of challenges which arise during the validation of interprocedural analysis results in an expandable object-oriented runtime environment. The approach abstracts from problem-specific properties and focuses on fundamental properties of any data flow analysis - namely the lattice representation of the data flow values and the monotony of transfer functions.

It is possible to observe the central principle of the validation approach which replaces the fix-point computation by a fix-point test several times in this generic model: The analysis resolves cyclic dependencies which stem from loop structures and from recursive method invocations while a linear pass is sufficient to check the corresponding result. Additionally, we also observe that several analyses, like the call graph construction and the type analysis for receiver types, can cyclically depend on each other, too. Such a kind of dependency requires that the analyser repeats the analyses several times until a common fix-point solution is reached. Again, the validator can avoid this iteration and is able to check the analyses results in a linear pass. Therefore, the approach in this thesis is only a first step to exploit the full potential of the validation of analysis results.

10.2 Future Directions

The discussion reveals several natural extension points, to increase the expressiveness of the framework.

Distributivity The summary function model restricts itself to distributive data flow problems. The advantage of distributive problems is that the precision of the result is independent to the sequence in which safe approximations and elementary transfer functions are applied. This ensures, that the intermediate results of the validation process do not depend on the way

the validator processes. Nevertheless, the general validation principle may also be applicable to non-distributive problems, if we synchronise the way in which analysis and validator process and normalise the analysis results.

Conservative Treatment of Nested Expressions The current implementation of the framework restricts the maximum nesting depth of expressions and safely approximates the parameter expressions if the nesting depth would exceed this limit. This strategy is safe and restricts the size of the summary functions, but it reduces the precision of the analysis. Essentially, the strategy restricts the number of subsequent elementary transfer functions and the maximum number of external method calls on a program path. The first restriction can be tackled if we take problem-specific properties of elementary transfer functions into account in a way which is similar to the compression of microtransformers in the graph-based approach of Reps. The second issue can be solved if we replace parameter expressions by references to intraprocedural summary functions and adopt the substitution mechanism in the interprocedural fix-point solver.

Program State The environment model works very well for variables on the call stack, because they cannot be modified by external calls. Thus, external function calls do not introduce a function variable expression for each local variable, but for the result of the method call only. Further extensions of the program state like data flow values in fields increase the number of function variable expressions the framework has to deal with. As a consequence it becomes even more important to replace the construction of nested parameter expressions by references to interprocedural summary functions as suggested in the previous paragraph. However, the number of summary functions the analysis approach has to store may increase to the number of invoke instructions in the program. The situation calls for an adopted version of the closed-program assumption where the analysis for example takes the visibility of fields into account in order to rule out external modifications immediately which would otherwise lead to additional function variable expressions.

Additional Analyses The generic structure of the framework calls for the specification of additional analyses. The most important candidate is the implementation of the type inference algorithm outlined in Section 7.3 because its result immediately improves the precision of the existing implementation for dynamic call resolution. The framework already meets the requirements of the algorithm. The specification of most instruction-level summary functions requires simple data flow functions which propagate the type information similar to copy constant propagation. Elementary transfer functions are required for the specification of array access instructions only, and they cannot lead to nested function expressions without breaking the type safety of the program. Therefore, we expect that the structure of the summary functions will remain linear in the size of the program state.

Safe Lower-Bounds The validator can derive a lower bound from an open summary function representation at any point in time by the substitution of variables with safe lower bounds for the summary functions or the variables in the invocation context which are represented by the variables. The most pessimistic element of the data flow problem exists always. However, it is also possible to derive more precise safe lower bounds for some data flow problems. For example, the declared type of a variable or of a result value can act as a safe lower bound in the type inference analysis. It is necessary to validate such a lower bound if we cannot make some specific assumptions about the program. Interestingly, the Java Bytecode Verification guarantees the correctness of the declared types by simple analysis of each method body. This observation is interesting because it shows that it is possible to use the results of a simpler analysis as a more precise lower bound in a more sophisticated analysis, for example to keep the size of the summary functions under control.

Alias and Points-To Analyses The analysis of the data flow via object fields requires at least a limited alias and points-to analysis, because the question which field is accessed by a read- or write-operation depends on the object reference used to access the field. Simple variants, which for example only identify accesses to the receiver reference of the call (`this`) are not much more complex than the type inference or copy constant propagation problems. In contrast, the validation of full fledged alias and points-to results may very well require additional extensions to the summary function model, because a straight-forward application of the existing modeling techniques can result in large program state representations and complex defining expressions. Nevertheless, it would be interesting to determine more precisely how far the current model is able to cope with this important class of analyses.

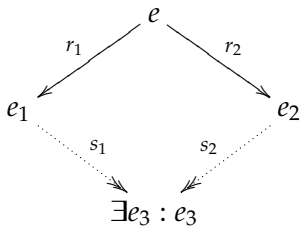
The investigation of the validation of analysis results in this thesis shows that the more complex interprocedural setting increases the potential of the proof-carrying code principle. The fundamental idea to replace a fix-point computation by a check of the fix-point solution applies to the investigation of recursive method invocations as well as to analyses which cyclically depend on each other.

Furthermore, the integration of function variables into the summary function model is a novel approach to represent results of separated software modules in a validatable way and the generic formulation of the model allows for a comparatively simple formulation of additional analyses. Thus, the validation of interprocedural analysis results forms an interesting basis for several directions of further research which have not been fully exploited yet.

Appendix

A Proofs

Proof 17 (Lemma 2) *A reduction relation $\rightarrow_E$ is locally confluent if the results e_1, e_2 of two different reduction steps r_1, r_2 are joinable - i.e. we can find two subsequent reduction sequences s_1 and s_2 which lead to the same expression e_3.*



We have to check that the property holds for each pair of the reductions $\xrightarrow{CF}$, $\xrightarrow{VAR}$, $\xrightarrow{BSC}$, $\xrightarrow{POUB}$ and $\xrightarrow{DSTR}$. Each of the cases contains several subcases which capture the different expression structures the reduction rules may be applicable in. Essentially, the subterms involved in the reduction rules can either be completely disjoint, share a common subterm or one term can be nested into the other.

The combinations which only consist of $\xrightarrow{CF}$, $\xrightarrow{VAR}$, and $\xrightarrow{BSC}$-reductions are easy to solve, so we are left with pairs of reductions that involve $\xrightarrow{POUB}$ or $\xrightarrow{DSTR}$-reductions. We assume once again that each function application expression takes a single parameter only, in order to simplify the notation. The $\xrightarrow{DSTR}$-reductions on function variable expressions are proven in the same way as the similar reductions on elementary function applications.

Throughout all proofs, let ub_{ij} denote the upper bound of a function application expression and let $c_i c_j$ denote the result of the conservative approximation $c_i \sqcap_L c_j$.
.

Reduction Pairs with $r_1 = POUB$

- $r_2 = CF$

1.

$$\begin{array}{lcl} e & = & \underline{t(p \sqcap c_1)} \sqcap \underline{c_2 \sqcap c_3} \\ e & \overset{POUB}{\longrightarrow} & t_1(p_1 \sqcap c_1) \sqcap c_2 ub_{11} \sqcap c_3 \\ & \overset{CF}{\longrightarrow} & t_1(p_1 \sqcap c_1) \sqcap c_3 c_2 ub_{11} \end{array}$$

$$\begin{array}{lcll} e & \overset{CF}{\longrightarrow} & t_1(p_1 \sqcap c_1) \sqcap c_2 c_3 & \\ & = & t_1(p_1 \sqcap c_1) \sqcap c_3 c_2 ub_{11} & \textit{if} \quad ub_{11} \sqsupseteq c_2 c_3 \\ & \overset{POUB}{\longrightarrow} & t_1(p_1 \sqcap c_1) \sqcap c_3 c_2 ub_{11} & \textit{else} \end{array}$$

2. *Obviously* $c_{ub} = [p_1 \sqcap c_1 \sqcap c_2]|_{[x:=\bot]} = [p_1 \sqcap c_2 c_3]|_{[x:=\bot]}$, *thus*

$$\begin{array}{lcl} e & = & \underline{t(p_1 \sqcap \underline{c_1 \sqcap c_2}) \sqcap c_3} \\ e & \overset{POUB}{\longrightarrow} & t_1(p_1 \sqcap c_2 \sqcap c_3) \sqcap ub_{11} \\ & \overset{CF}{\longrightarrow} & t_1(p_1 \sqcap c_2 c_3) \sqcap ub_{11} \end{array}$$

$$\begin{array}{lcl} e & \overset{CF}{\longrightarrow} & t_1(p_1 \sqcap c_2 c_3) \sqcap c_3 \\ & \overset{POUB}{\longrightarrow} & t_1(p_1 \sqcap c_2 c_3) \sqcap ub_{11} \end{array}$$

- $r_2 = VAR$ *Similar to CF*
- $r_2 = BSC$

1.

$$\begin{array}{lcl} e & = & \underline{t_1(p_1 \sqcap c_1) \sqcap \underline{c_2 \sqcap \bot}} \\ e & \overset{POUB}{\longrightarrow} & t_1(p_1 \sqcap c_1) \sqcap c_1 ub_{11} \sqcap \bot \\ & \overset{BSC}{\longrightarrow} & t_1(p_1 \sqcap c_1) \sqcap \bot \\ & \overset{BSC}{\longrightarrow} & \bot \end{array}$$

$$\begin{array}{lcl} e & \overset{BSC}{\longrightarrow} & t_1(p_1 \sqcap c_1) \sqcap \bot \\ & \overset{BSC}{\longrightarrow} & \bot \end{array}$$

2.

$$e = \underline{\underline{t_1(p_1 \sqcap c_1) \sqcap \bot}}$$

$\overset{POUB}{\longrightarrow}$ *not applicable because* $\nexists c \sqsubset \bot$.

3. *Obviously* $[p_1 \sqcap \bot]|_{[x:=\bot]} = \bot = [\bot]|_{[x:=\bot]}$, *thus*

$$\begin{array}{lll} e & = & \underline{t_1(\underline{p_1 \sqcap \bot}) \sqcap c_2} \\ e & \overset{POUB}{\longrightarrow} & t_1(p_1 \sqcap \bot) \sqcap c_2 ub_{11} \\ & \overset{BSC}{\longrightarrow} & t_1(\bot) \sqcap c_2 ub_{11} \end{array}$$

$$\begin{array}{lll} e & \overset{BSC}{\longrightarrow} & t_1(\bot) \sqcap c_2 \\ & \overset{POUB}{\longrightarrow} & t_1(\bot) \sqcap c_2 ub_{11} \end{array}$$

- $r_2 = POUB$
  1. *Let* $ub_1 = t_1(p_1|_{[x:=\bot]} \sqcap c_1$ *and* $ub_2 = t_2(p_2|_{[x:=\bot]} \sqcap c_2$

$$\begin{array}{lll} e & = & \underline{t_1(p_1 \sqcap c_1)} \sqcap c \sqcap \underline{t_2(p_2 \sqcap c_2)} \\ e & \overset{POUB}{\longrightarrow}_1 & t_1(p_1 \sqcap c_1) \sqcap cub_1 \sqcap t_2(p_2 \sqcap c_2) \\ & \overset{POUB}{\longrightarrow}_2 & t_1(p_1 \sqcap c_1) \sqcap cub_1 ub_2 \sqcap t_2(p_2 \sqcap c_2) \qquad \textit{if} \quad cub_1 ub_2 \sqsubset cub_1 \\ & = & t_1(p_1 \sqcap c_1) \sqcap cub_1 ub_2 \sqcap t_2(p_2 \sqcap c_2) \qquad \textit{else} \end{array}$$

$$e \overset{POUB}{\longrightarrow}_2 \qquad \textit{anagolous}$$

  2. *Let* $ub_2 = t_2(p_2|_{[x:=\bot]} \sqcap c_2$ *and* $ub_1 = t_1(ub_2 \sqcap c_1)$*:*

$$\begin{array}{lll} e & = & \underline{t_1(\underline{t_2(p_2 \sqcap c_2) \sqcap c_1}) \sqcap c_3} \\ e & \overset{POUB}{\longrightarrow}_1 & t_1(t_2(p_2 \sqcap c_2) \sqcap c_1) \sqcap c_3 ub_1 \\ & \overset{POUB}{\longrightarrow}_2 & t_1(t_2(p_2 \sqcap c_2) \sqcap c_1 ub_2) \sqcap c_3 ub_1 \end{array}$$

$$\begin{array}{lll} e & \overset{POUB}{\longrightarrow}_2 & t_1(t_2(p_2 \sqcap c_2) \sqcap c_1 ub_2) \sqcap c_3 \\ & \overset{POUB}{\longrightarrow}_1 & t_1(t_2(p_2 \sqcap c_2) \sqcap c_1 ub_2) \sqcap c_3 ub_1 \\ & \textit{because} & t_2([p_2]|_{[x:=\bot]} \sqcap c_1 ub_2) \\ & & = [ub_2 \sqcap c_1 ub_2]|_{[x:=\bot]} \\ & & = [ub_2 \sqcap c_1]|_{[x:=ub_2 \sqcap c_1]} \\ & \Rightarrow & t_1([t_2(p_2 \sqcap c_2) \sqcap c_1 ub_2]|_{[x:=\bot]}) = ub_1 \end{array}$$

- $r_2 = DSTR$

1.

$$
\begin{array}{rcl}
e & = & \underline{t_1(p_1 \sqcap c_1) \sqcap t_1(p_2 \sqcap c_2)} \sqcap \underline{t_3(p_3 \sqcap c_3) \sqcap c_4} \\
e & \overset{DSTR}{\longrightarrow} & t_1(p_1 \sqcap c_1 \sqcap p_2 \sqcap c_2) \sqcap t_3(p_3 \sqcap c_2) \sqcap c_4 \\
 & \overset{POUB}{\longrightarrow} & t_1(p_1 \sqcap c_1 \sqcap p_2 \sqcap c_2) \sqcap t_3(p_3 \sqcap c_2) \sqcap c_4 ub_3
\end{array}
$$

$$
\begin{array}{rcl}
e & \overset{POUB}{\longrightarrow} & t_1(p_1 \sqcap c_1) \sqcap t_1(p_2 \sqcap c_2) \sqcap t_3(p_3 \sqcap c_2) \sqcap c_4 ub_3 \\
 & \overset{DSTR}{\longrightarrow} & t_1(p_1 \sqcap c_1 \sqcap p_2 \sqcap c_2) \sqcap t_3(p_3 \sqcap c_2) \sqcap c_4 ub_3
\end{array}
$$

2.

$$
\begin{array}{rcl}
e & = & \underline{t_1(p_1 \sqcap c_1) \sqcap \underline{t_1(p_2 \sqcap c_2)} \sqcap c_3} \\
e & \overset{DSTR}{\longrightarrow} & t_1(p_1 \sqcap c_1 \sqcap p_2 \sqcap c_2) \sqcap c_3 \\
 & \overset{POUB}{\longrightarrow} & t_1(p_1 \sqcap c_1 \sqcap p_2 \sqcap c_2) \sqcap c_3 ub_2 \\
 & \textit{because} & p_1 \sqcap c_1 \sqcap p_2 \sqcap c_2 \sqsubseteq p_2 \sqcap c_2 \textit{ due to the semantics of } \sqcap \\
 & \Rightarrow & t_1(p_1 \sqcap c_1 \sqcap p_2 \sqcap c_2) \sqsubseteq t_1(p_2 \sqcap c_2) \\
 & & \textit{due to the monotony of } \quad t_1 \\
 & \Rightarrow & \textit{POUB is applicable because} \\
 & & \textit{POUB was applicable for } \quad t_1(p_2 \sqcap c_2)
\end{array}
$$

$$
\begin{array}{rcl}
e & \overset{POUB}{\longrightarrow} & t_1(p_1 \sqcap c_1) \sqcap t_1(p_2 \sqcap c_2) \sqcap c_3 ub_2 \\
 & \overset{DSTR}{\longrightarrow} & t_1(p_1 \sqcap c_1 \sqcap p_2 \sqcap c_2) \sqcap c_3 ub_2
\end{array}
$$

If $\overset{POUB}{\longrightarrow}$ is applicable to $t_1(p_1 \sqcap c_1)$ as well then it can either be applied, or it is subsumed by an conservative approximation of the parameter expression after application of $\overset{DSTR}{\longrightarrow}$. This requires distributivity of t.

3.

$$
\begin{array}{rcl}
e & = & \underline{t_1(\underline{t_3(p_3 \sqcap c_3) \sqcap c_1}) \sqcap t_1(p_2 \sqcap c_2)} \\
e & \overset{POUB}{\longrightarrow} & t_1(t_3(p_3 \sqcap c_3) \sqcap c_1 ub_3) \sqcap t_1(p_2 \sqcap c_2) \\
 & \overset{DSTR}{\longrightarrow} & t_1(t_3(p_3 \sqcap c_3) \sqcap c_1 ub_3 \sqcap p_2 \sqcap c_2)
\end{array}
$$

$$
\begin{array}{rcl}
e & \overset{DSTR}{\longrightarrow} & t_1(t_3(p_3 \sqcap c_3) \sqcap c_1 \sqcap p_2 \sqcap c_2) \\
 & \overset{POUB}{\longrightarrow} & t_1(t_3(p_3 \sqcap c_3) \sqcap c_1 ub_3 \sqcap p_2 \sqcap c_2)
\end{array}
$$

Reduction Pairs with $r_1 = DSTR$

- $r_2 = CF$

1.

$$\begin{aligned} e &= \underline{c_1 \sqcap c_2} \sqcap \underline{t_1(p_1) \sqcap t_1(p_2)} \\ e &\xrightarrow{CF} c_1c_2 \sqcap t_1(p_1) \sqcap t_2(p_2) \\ &\xrightarrow{DSTR} c_1c_2 \sqcap t_1(p_1 \sqcap p_2) \end{aligned}$$

$$\begin{aligned} &\xrightarrow{DSTR} c_1 \sqcap c_2 \sqcap t_1(p_1 \sqcap p_2) \\ &\xrightarrow{CF} c_1c_2 \sqcap t_1(p_1 \sqcap p_2) \end{aligned}$$

2.

$$\begin{aligned} e &= \underline{t_1(p_1 \sqcap \underline{c_1 \sqcap c_2}) \sqcap t_1(p_2)} \\ e &\xrightarrow{CF} t_1(p_1 \sqcap c_1c_2) \sqcap t_1(p_2) \\ &\xrightarrow{DSTR} t_1(p_1 \sqcap c_1c_2 \sqcap p_2) \end{aligned}$$

$$\begin{aligned} e &\xrightarrow{DSTR} t_1(p_1 \sqcap c_1 \sqcap c_2 \sqcap p_2) \\ &\xrightarrow{CF} t_1(p_1 \sqcap c_1c_2 \sqcap p_2) \end{aligned}$$

- $r_2 = VAR$ *Similar to* $r_2 = CF$
- $r_2 = BSC$

1.

$$\begin{aligned} e &= \underline{t_1(p_1) \sqcap t_1(p_2)\underline{\sqcap \bot}} \\ e &\xrightarrow{BSC} t_1(p_1) \sqcap \bot \\ &\xrightarrow{BSC} \bot \end{aligned}$$

$$\begin{aligned} e &\xrightarrow{DSTR} t_1(p_1 \sqcap p_2) \sqcap \bot \\ &\xrightarrow{BSC} \bot \end{aligned}$$

2.

$$\begin{aligned} e &= \underline{t_1(\underline{p_1 \sqcap \bot}) \sqcap t_1(p_2)} \\ e &\xrightarrow{BSC} t_1(\bot) \sqcap t_1(p_2) \\ &\xrightarrow{DSTR} t_1(\bot \sqcap p_2) \\ &\xrightarrow{BSC} t_1(\bot) \end{aligned}$$

$$\begin{aligned} &\xrightarrow{DSTR} t_1(p_1 \sqcap \bot \sqcap p_2) \\ &\xrightarrow{BSC} t_1(\bot p_2) \\ &\xrightarrow{BSC} t_1(\bot) \end{aligned}$$

- $r_2 = DSTR$

  1.

$$\begin{array}{lll} e & = & \underline{t_1(p_1) \sqcap t_1(p_2)} \sqcap \underline{t_2(p_3) \sqcap t_2(p_4)} \\ e & \overset{DSTR}{\longrightarrow}_1 & t(p_1 \sqcap p_2) \sqcap t_2(p_3) \sqcap t_2(p_4) \\ & \overset{DSTR}{\longrightarrow}_2 & t(p_1 \sqcap p_2) \sqcap t_2(p_3 \sqcap p_4) \end{array}$$

$$e \overset{DSTR}{\longrightarrow}_2 \quad \textit{analogous}$$

  2.

$$\begin{array}{lll} e & = & \underline{t_1(p_1) \sqcap t_1(p_2) \sqcap t_1(p_3)} \\ e & \overset{DSTR}{\longrightarrow}_1 & t_1(p_1 \sqcap p_2) \sqcap t_1(p_3) \\ & \overset{DSTR}{\longrightarrow} & t_1(p_1 \sqcap p_2 \sqcap p_3) \end{array}$$

$$e \overset{DSTR}{\longrightarrow}_2 \quad \textit{analogous}$$

  3.

$$\begin{array}{lll} e & = & \underline{t_1(\underline{t_2(p_2) \sqcap t_2(p_3)}) \sqcap t_1(p_1)} \\ e & \overset{DSTR}{\longrightarrow}_1 & t_1(t_2(p_2) \sqcap t_2(p_3) \sqcap p_1) \\ & \overset{DSTR}{\longrightarrow}_2 & t_1(t_2(p_2 \sqcap p_3) \sqcap p_1) \end{array}$$

$$\begin{array}{lll} e & \overset{DSTR}{\longrightarrow}_2 & t_1(t_2(p_2 \sqcap p_3)) \sqcap p_1 \\ & \overset{DSTR}{\longrightarrow}_1 & t_1(t_2(p_2 \sqcap p_3) \sqcap p_1) \end{array}$$

Proof 18 (Theorem 5) *The following proof is the full version which is extended by function variable expressions which are introduced in Section 5.4.2.*

Firstly, we have to prove that e_3 is weaker or equal to e_1 and e_2 with respect to $\sqsubseteq_{E\downarrow}$. Secondly, we show that e_3 is maximal.

1. $e_3 \sqsubseteq_{E\downarrow} e_1$*:*

$$\begin{array}{lll} \textit{Let} & e_1\downarrow & = \bigsqcap_{i \in TI_1} t_i(p1_i) \sqcap \bigsqcap_{j \in SJ_1} s_j(q1_j) \bigsqcap_{k \in VK_1} x_k \sqcap c_1 \\ \textit{and} & e_2\downarrow & = \bigsqcap_{i \in TI_2} t_i(p2_i) \sqcap \bigsqcap_{j \in SJ_2} s_j(q2_j) \bigsqcap_{k \in VK_2} x_k \sqcap c_2 \end{array}$$

then

$$
\begin{aligned}
e_3\downarrow \quad &= \quad e_1\downarrow \sqcap e_2\downarrow \\
&= \quad [\prod_{i'\in TI_1-TI_2} t_{i'}(p1_{i'}) \sqcap \prod_{i''\in TI_2-TI_1} t_{i''}(p2_{i''}) \sqcap \\
&\qquad \prod_{i'''\in TI_1\cap TI_2} t_{i'''}(p1_{i'''}) \sqcap t_{i'''}(p2_{i'''}) \sqcap \\
&\qquad \prod_{j'\in SJ_1-SJ_2} s_{j'}(p1_{j'}) \sqcap \prod_{j''\in SJ_2-SJ_1} s_{j''}(p2_{j''}) \sqcap \\
&\qquad \prod_{j'''\in SJ_1\cap SJ_2} s_{j'''}(p1_{j'''} \sqcap s_{j'''}(p2_{j'''}) \sqcap \\
&\qquad \prod_{k'\in VK_1-VK_2} x_{k'} \sqcap \prod_{k''\in VK_2-VK_1} x_{k''} \sqcap \prod_{k'''\in VK_1\cap VK_2} x_{k'''} \sqcap x_{k'''} \\
&\qquad c_1 \sqcap c_2]\downarrow \\
\xrightarrow{DSTR} * \quad &[\prod_{i'\in TI_1-TI_2} t_{i'}(p1_{i'}) \sqcap \prod_{i''\in TI_2-TI_1} t_{i''}(p2_{i''}) \sqcap \prod_{i'''\in TI_1\cap TI_2} t_{i'''}(p1_{i'''} \sqcap p2_{i'''}) \sqcap \\
&\qquad \prod_{j'\in SJ_1-SJ_2} s_{j'}(p1_{j'}) \sqcap \prod_{j''\in SJ_2-SJ_1} s_{j''}(p2_{j''}) \sqcap \prod_{j'''\in SJ_1\cap SJ_2} s_{j'''}(p1_{j'''} \sqcap p2_{j'''}) \sqcap \\
&\qquad \prod_{k'\in VK_1-VK_2} x_{k'} \sqcap \prod_{k''\in VK_2-VK_1} x_{k''} \sqcap \prod_{k'''\in VK_1\cap VK_2} x_{k'''} \sqcap x_{k'''} \\
&\qquad c_1 \sqcap c_2]\downarrow \\
\xrightarrow{VAR} * \quad &[\prod_{i'\in TI_1-TI_2} t_{i'}(p1_{i'}) \sqcap \prod_{i''\in TI_2-TI_1} t_{i''}(p2_{i''}) \sqcap \prod_{i'''\in TI_1\cap TI_2} t_{i'''}(p1_{i'''} \sqcap p2_{i'''}) \sqcap \\
&\qquad \prod_{j'\in SJ_1-SJ_2} s_{j'}(p1_{j'}) \sqcap \prod_{j''\in SJ_2-SJ_1} s_{j''}(p2_{j''}) \sqcap \prod_{j'''\in SJ_1\cap SJ_2} s_{j'''}(p1_{j'''} \sqcap p2_{j'''}) \sqcap \\
&\qquad \prod_{k'''\in VK_1\cup VK_2} x_{k'''} \\
&\qquad c_1 \sqcap c_2]\downarrow \\
\xrightarrow{CF} \quad &[\prod_{i'\in TI_1-TI_2} t_{i'}(p1_{i'}) \sqcap \prod_{i''\in TI_2-TI_1} t_{i''}(p2_{i''}) \sqcap \prod_{i'''\in TI_1\cap TI_2} t_{i'''}(p1_{i'''} \sqcap p2_{i'''}) \sqcap \\
&\qquad \prod_{j'\in SJ_1-SJ_2} s_{j'}(p1_{j'}) \sqcap \prod_{j''\in SJ_2-SJ_1} s_{j''}(p2_{j''}) \sqcap \prod_{j'''\in SJ_1\cap SJ_2} s_{j'''}(p1_{j'''} \sqcap p2_{j'''}) \sqcap \\
&\qquad \prod_{k'''\in VK_1\cup VK_2} x_{k'''} \\
&\qquad c_1c_2]\downarrow \\
\sqsubseteq_{E\downarrow} \quad &e_1\downarrow
\end{aligned}
$$

Clearly, the definition of $\sqsubseteq_{E\downarrow}$ holds, because $e\downarrow$ either contains at least the same subexpressions or application expression with weaker parameter expressions. If $c_1c_2 = \bot$ than the expression reduces further to $\bot$ and the proposition also holds.

2. $e_3 \sqsubseteq_{E\downarrow} e_2$*: Analogous.*
3. e_3 *is maximal with respect to* $\sqsubseteq_{E\downarrow}$*: Assume* $\exists e_4 : e_4 \sqsubset_{E\downarrow} e_3 \wedge e_4 \sqsubseteq_{E\downarrow} e_1 \wedge e_4 \sqsubseteq_{E\downarrow} e_2$*.*

Due to $e_4 \sqsupset_{E\downarrow} e_3$ *one of the following conditions holds:*

a) There is a subexpression se in $e_3\downarrow$ *which does not exist in* $e_4\downarrow$.
This expression has to occur in either e_1 *or in* e_2 *and in turn has to occur in* $e_4\downarrow$ *due to the fact that* $e_4 \sqsubseteq_{E\downarrow} e_1 \wedge e_4 \sqsubseteq_{E\downarrow} e_2$.

b) If $e_3\downarrow$ *and* $e_4\downarrow$ *contain all the same subexpressions, then there must be a function application expression which has a weaker parameter expression in* e_3 *than in* e_4. *This cannot be the case due to the maximality of* $p_1 \sqcap_L p_2$ *in L and due to the induction hypothesis on expressions* p_1 *and* p_2 *with smaller depth than* e_4.

c) If $e_4\downarrow \perp$, *than* $\nexists e_3 : e_4 \sqsupset e_3$.

Bibliography

[AAP06] Elvira Albert, Puri Arenas, and Germán Puebla. An incremental approach to abstraction-carrying code. In *LPAR*, pages 377–391, 2006.

[AASPH06] Elvira Albert, Puri Arenas-Sánchez, Germán Puebla, and Manuel V. Hermenegildo. Reduced certificates for abstraction-carrying code. In *ICLP*, pages 163–178, 2006.

[AC76] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137–147, 1976.

[ADvRF01] Amme, Dalton, von Ronne, and Franz. SafeTSA: A Type Safe and Referentially Secure Mobile-Code Representation Based on Static Single Assignment Form. In *SIGPLAN'01 Conference on Programming Language Design and Implementation*, pages 137–147, 2001.

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson, Addison Wesley, 2nd edition edition, 2007.

[AM95] Martin Alt and Florian Martin. Generation of efficient interprocedural analyzers with PAG. In *SAS'95, Static Analysis Symposium*, volume 983, pages 33–50, 1995.

[Amm07] Amme. Data flow analysis as a general concept for the transport of verifiable program annotations. In *Proceedings of the 5th International Workshop on Compiler Optimization meets Compiler Verification (COCV 2006)*, volume 176 of *Electronic Notes in Theoretical Computer Science*, pages 97–108, 2007.

[And94] Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[APH04] Elvira Albert, Germán Puebla, and Manuel V. Hermenegildo. Abstraction-carrying code. In *LPAR*, pages 380–397, 2004.

[APH05] Elvira Albert, Germán Puebla, and Manuel V. Hermenegildo. An abstract interpretation-based approach to mobile code safety. *Electr. Notes Theor. Comput. Sci.*, 132(1):113–129, 2005.

[BCT94] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[Bel66] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[BFM06] Cinzia Bernardeschi, Nicoletta De Francesco, and Luca Martini. Using postdomination to reduce space requirements of data flow analysis. *Inform. Process. Lett.*, 98(1):11–18, 2006.

[BGR05] G. Balakrishnan, R. Gruian, and T. Reps. CodeSurfer/x86: A platform for analyzing x86 executables. In R. Bodik, editor, *CC 2005*, volume 3443, pages 250–254. Springer-Verlag, 2005.

[BLMM05] Cinzia Bernardeschi, Giuseppe Lettieri, Luca Martini, and Paolo Masci. A space-aware bytecode verifier for java cards. In *Proceedings of the First Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode 2005)*, volume 141 of *Electronic Notes in Theoretical Computer Science*, pages 237–254, 2005.

[BLTY03] Gilad Bracha, Tim Lindholm, Wei Tao, and Frank Yellin. *CLDC Byte Code Typechecker Specification*. SUN Microsystems, January 2003.

[BMA03] Pierre-Luc Brunelle, Ettore Merlo, and Giuliano Antoniol. Investigating java type analyses for the receiver-classes testing criterion. In *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, page 419, 2003.

[BR01] Thomas Ball and Srinam Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM Press, 2001.

[BR02] Thomas Ball and Sriram K. Rajamani. The slam project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.

[BS96] David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 324–341, New York, NY, USA, 1996. ACM Press.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CC02] Patrick Cousot and Radhia Cousot. Modular static analysis. In Horspool, editor, *Proceedings of the Eleventh International Conference on Compiler Construction (CC 2002)*, pages 159–178. LLNCS 2304, Springer, Berlin, April 6—14 2002.

[Cha82] Gregory J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–101. ACM Press, 1982.

[CHK04] Keith Cooper, Timothy Harvey, and Ken Kennedy. Iterative dataflow analysis revisited. Technical Report TR04-100, Rice University, 2004.

[Cor] Standard Performance Evaluation Corporation. Spec jvm98. http://www.spec.org/benchmarks.html.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP 94 Bologna, Italy, July 4, 1994 Proceedings*, pages 77–101, London, UK, 1995. Springer-Verlag.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[FL88] Charles N. Fisher and Richard J. LeBlanc. *Crafting a Compiler*. Benjamin/Cummings Publishing Company, 1988.

[GR07] Denis Gopan and Thomas Reps. Low-level library analysis and summarization. In *19th International Conference, CAV 2007, Berlin*, volume 4590 of *LNCS*, pages 68–81. Springer-Verlag, 2007.

[Gro98] David Paul Grove. *Effective Interprocedural Optimization of Object-Oriented Languages*. PhD thesis, University of Washington, 1998.

[GT07] Sumit Gulwani and Ashish Tiwari. Computing procedure summaries for interprocedural analysis. In De Nicola, editor, *European Symposium on Programming, ESOP 2007*, volume 4421 of *LNCS*, pages 253–267, 2007.

[GTN04] Sumit Gulwani, Ashish Tiwari, and George C. Necula. Join algorithms for the theory of uninterpreted functions. In *24th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 3328 of *LNCS*, pages 311–323. Springer-Verlag, 2004.

[GW76] Susan L. Graham and Mark Wegman. A fast and usually linear algorithm for global flow analysis. *J. ACM*, 23(1):172–202, 1976.

[Hec77] Hecht. *Flow Analysis of Computer Programs*. Elsevier, 1977.

[HU74] M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *J. ACM*, 21(3):367–375, 1974.

[HU75] Matthew S. Hecht and Jeffrey D. Ullman. A simple algorithm for global data flow analysis problems. *SIAM Journal on Computing*, 4(4):519–532, 1975.

[JM81] Neil D. Jones and Steven S. Muchnick. *Program Flow Analysis*, chapter Complexity of Flow Analysis, Inductive Assertions, and a Language Due to Dijkstra, pages 380–393. Prentice Hall, 1981.

[Kil73] Gary Kildall. A unified approach to global program optimization. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM Press, 1973.

[KK05] Karsten Klohs and Uwe Kastens. Memory requirements of java bytecode verification on limited devices. *Electr. Notes Theor. Comput. Sci.*, 132(1):95–111, 2005.

[Kno99] Jens Knoop. *Optimal Interprocedural Program Optimization: A New Framework and Its Application*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[KU76] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *J. ACM*, 23(1):158–171, 1976.

[KU77] J.B. Kam and J.D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[LH03] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.

[Lho06] Ondřej Lhoták. *Program Analysis Using Binary Decision Diagrams*. PhD thesis, McGill University, Montreal, 2006.

[LR08] Junghee Lim and Thomas Reps. A system for generating static analyzers for machine instructions. In Laurie Hendren, editor, *Proceedings of the 17th International Conference on Compiler Construction*, volume 4959/2008, pages 36–52. Springer-Verlag, 2008.

[MORS05] Markus Müller-Olm, Oliver Rüthing, and Helmut Seidl. Checking herbrand equalities and beyond. In Radhia Cousot, editor, *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France*, volume 3385, pages 79–96, 2005.

[MOS04] Markus Müller-Olm and Helmut Seidl. A note on Karr's algorithm. In *Automata, Languages and Programming*, volume 3142/2004, pages 1016–1028, 2004.

[MOSS99] Markus Müller-Olm, David A. Schmidt, and Bernhard Steffen. Model-checking: A tutorial introduction. In *SAS '99: Proceedings of the 6th International Symposium on Static Analysis*, volume 1694, pages 330–354, 1999.

[MR90] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks: a unified model. *Acta Inf.*, 28(2):121–163, 1990.

[Muc97] Steven Muchnick. *Advanced Compiler Design and Implementation*. Academic Press, 1997.

[Nec97] George C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM Press.

[New42] M.H.A. Newman. On theories with a combinatorial definition of equivalence. In *Annals of Mathematics*, volume 43, pages 223–243. Princeton University, 1942.

[NL96] Necula and Lee. Safe Kernel Extensions without Run-Time Checking. In *Second Symposium on Operating Systems Design and Implementations*. USENIX, 1996.

[PACJ+08] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, July 22–24 2008.

[RH07] Venkatesh Prasad Ranganath and John Hatcliff. Slicing concurrent java programs using indus and kaveri. *International Journal on Software Tools for Technology Transfer (STTT)*, 9:489–504, 2007.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, New York, NY, USA, 1995. ACM Press.

[RKM06] Atanas Rountev, Scott Kagan, and Thomas Marlowe. Interprocedural dataflow analysis in the presence of large libraries. In *: Compiler Construction*, pages 2–16. Springer-Verlag, 2006.

[RKS99] Oliver Rüthing, Jens Knoop, and Bernhard Steffen. Detecting equalities of variables: Combining efficiency with precision. In *SAS '99: Proceedings of the 6th International Symposium on Static Analysis*, volume 1694, pages 232–247. Springer-Verlag, 1999.

[RMR03] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Fragment class analysis for testing of polymorphism in java software. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 210–220. IEEE Computer Society, 2003.

[Ros03] Eva Rose. Lightweight bytecode verification. *J. Autom. Reason.*, 31(3-4):303–334, November 2003.

[Rou02] Antanas Rountev. *Dataflow Analysis of Software Fragments*. PhD thesis, Rutgers University, aug 2002.

[Rou05] Atanas Rountev. Component-level dataflow analysis. In *Component-Based Software Engeneering*, pages 82–89, 2005.

[RP86] Barbara G. Ryder and Marvin C. Paull. Elimination algorithms for data flow analysis. *ACM Comput. Surv.*, 18(3):277–316, 1986.

[RR98] E. Rose and K. H. Rose. Lightweight Bytecode Verification. In *Workshop "Formal Underpinnings of the Java Paradigm", OOPSLA'98*, 1998.

[RR01] Atanas Rountev and Barabara Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 20–36. Springer-Verlag, 2001.

[RSX08] Atanas Rountev, Mariana Sharp, and Guoqing Xu. Ide dataflow analysis in the presence of large object-oriented libraries. In Laurie Hendren, editor, *Proceedings of the 17th International Conference on Compiler Construction*, volume 4959/2008, pages 53–68. Springer-Verlag, 2008.

[Sch98] David A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 38–48. ACM, 1998.

[SP81] Micha Sharir and Amir Pnueli. *Program Flow Analysis*, chapter Two Approaches to Interprocedural Data Flow Analysis, pages 189–233. Prentice Hall, 1981.

[SRH96] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *TAPSOFT '95: Selected papers from the 6th international joint conference on Theory and practice of software development*, pages 131–170, Amsterdam, The Netherlands, 1996. Elsevier Science Publishers B. V.

[SS98] David Schmidt and Bernhard Steffen. Program analysis as model checking of abstract interpretations. In *SAS '98: Proceedings of the 5th International Symposium on Static Analysis*, volume 1503, pages 351–380. Springer-Verlag, 1998.

[Ste91] Bernhard Steffen. Data flow analysis as model checking. In *TACS '91: Proceedings of the International Conference on Theoretical Aspects of Computer Software*, volume 526/1991, pages 346–365, London, UK, 1991. Springer-Verlag.

[Tar81] Robert Endre Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28:594–614, 1981.

[Thi02] Michael Thies. *"Combining Static Analysis of Java Libraries with Dynamic Optimization"*. PhD thesis, University Paderborn, 2002.

[TP00] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. *SIGPLAN Not.*, 35(10):281–293, 2000.

[vR05] Jeffery von Ronne. *A Safe and Efficient Machine-Independent Code Transportation Format Based on Static Single Assignment Form and Applied to Just-In-Time Compilation*. PhD thesis, University of California, Irvine, 2005.

[VRCG+99] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, 1999.

[Wei81] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. *SIGPLAN Not.*, 34(10):187–206, 1999.

[YYC08] Greta Yorsh, Eran Yahav, and Satish Chandra. Generating precise and concise procedure summaries. In *POPL '08: Proceedings of*

the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 221–234. ACM Press, 2008.