Generating interpreters from compiler specifications

Uwe Kastens

Fachbereich Mathematik/Informatik
Universität–GH–Paderborn
Warburger Str. 100
D - 33098 Paderborn

Bericht tr-ri-94-151
Reihe Informatik
August 1994

Generating Interpreters from Compiler Specifications

Uwe Kastens, University of Paderborn, FRG

Abstract

The semantics of a programming language is described by a mapping from the source language constructs to operations and data of an abstract machine. Such mappings are implemented by compiler frontends. They can be effectively generated from specifications using toolsets like Eli. In this paper it is shown that efficient interpreters can be generated from the same specifications by simply adding an implementation of the source machine. The technique is based on iterative tree walking evaluators generated from attribute grammar specifications.

1 Introduction

The definition of a programming language describes the notation of programs and the effects they yield if executed on some input. Constraints are stated for a program and its execution being well-defined. The effects of program execution are usually described in terms of an execution model close to the source language concepts, an "abstract source language machine". The language definition then maps language constructs onto data and storage entities and operation sequences of that machine.

The abstract source language machine separates the structural properties and static semantics of the language from its dynamic semantics. Common compiler technique follows this separation: A compiler frontend translates the source language into an intermediate language, which is translated to target machine code by a compiler backend. Execution of the target program must have the same effect as described for executing the intermediate program.

A source language can also be implemented by an interpreter. Some languages are designed for being interpreted rather than compiled. They usually have a simple syntactic structure and an almost 1:1 mapping to their execution models. Examples are LISP, APL or command languages. If the language is designed for compilation it may be useful, too, to implement it by an interpreter: Programs are executable without having a compiler backend. Thus an interpreter may support rapid prototyping for language design, source level program debugging, or validation of compiler frontends and backends.

An interpreter implementation has to perform — at least conceptually — the same mapping from the source language to the abstract source language machine as the compiler does. Hence implementation effort is saved if that part of the implementation can be reused.

In this paper we demonstrate how interpreters and compilers can be generated from the same specification using Eli [2,15], a powerful toolset for language implementation. The specification of the mapping from the source language to an abstract machine is well supported by this toolset. Such a specification yields a compiler frontend. If an implementation of the abstract machine is added, that specification can also produce an interpreter which executes operations rather than emitting them as intermediate code.

Our approach is based on the following concepts: The abstract source machine is described by a set of functions. They operate on data structures thus maintaining the machines state according to its data model. The abstract tree of the program is mapped to a sequence of machine function calls. That mapping is specified in terms of an attribute grammar [6,9]. Calls of the machine functions are associated to tree node contexts. Sequences of such instructions are composed according to the tree structure and to attribute dependencies which specify pre- and postconditions of the instructions. So far this is conventional compiler technique. Instead of emitting source machine instructions they are immediately executed. Hence the instruction memory of a conventional interpreter is replaced by the tree walk algorithm that is generated as attribute evaluator to produce the instructions. Arbitrary branching in the dynamic control flow is achieved by iterating certain parts of the tree walk. In [11] a straight-forward extension of attribute grammar by iteration specification and a technique for evaluator generation is described. It is implemented in the LIGA System [6]. This technique of evaluator generation from dependencies of computations automatically determines the tree walk part to be iterated. Hence, it separates the computations for static semantics from those for dynamic semantics which may be iterated for interpretation.

The topic of integrated compiler and interpreter generation has been investigated since the midth of the seventies. Most of the semantics-directed compiler generators [10,12] are based on the method of denotational semantics [14]. A denotational specification describes a mapping from the abstract syntax to a functional expression. Its evaluation yields the effect of program execution. An interpreter can be obtained immediately by implementing the description in a functional programming language. But it is more difficult to obtain a compiler and an interpreter separately. The problem of distinction between static and dynamic semantics is attacked by applying the method of partial evaluation to the denotational description [3]. The difficulty of this strategy lies in the need to recover rather simple and well-understood translation concepts from rather complex descriptions on a very basic level, e. g. control flow translation from continuation semantics or name analysis from the use of identifier mapping functions. The same problems apply to the dynamic semantics: Concepts like runtime stack organization have to be recovered from patterns of location function use. It is clear that such a general method for language implementation can not achieve the efficiency of compilers and interpreters which are derived from specifications that identify such high-level concepts directly. More recent approaches in semantics directed language specification and implementation, like action semantics [13] integrate such high-level concepts by mapping to a set of actions which can be understood as operations of an abstract machine.

Structuring	Lexical analysis	Scanning Conversion
	Syntactic analysis	Parsing Tree construction
Translation	Semantic analysis	Name analysis Type analysis
	Transformation	Data mapping Action mapping
Encoding	Code generation	Execution-order determination Register allocation Instruction selection
	Assembly	Instruction encoding Internal address resolution External address resolution

Figure 1: Compilation Subproblems

2 Problem Decomposition

Many years of research and experience in Compiler Construction led to a generally accepted model for decomposition of compilation tasks, as shown in Figure 1 taken form [2].

Lexical analysis transforms the input program representation from a stream of characters into a stream of tokens, e. g. identifiers, numbers, operators, and keywords. It determines the role of each token within the program structure, and stores and encodes values associated with tokens. Syntactic analysis determines the program structure by parsing the token stream and represents the result in form of a tree. Each node together with its subtree is an abstraction of an occurrence of a language construct in the input program, e. g. a declaration, statement, or expression.

All tasks of the subsequent translation phase are expressed by computations associated to nodes of that abstract program tree. The name analysis tasks maps occurrences of identifiers to internal representations of named objects, e. g. variables, parameters, or record fields, according to the scope rules of the language. Type analysis associates the type property to objects and expression nodes and validates typing restrictions of the language.

The transformation task maps the data objects of the program onto the storage of an abstract machine, and transforms statements and operations into an instruction sequence for that machine. For programming language compilers an abstract machine model is usually defined by an intermediate language. It serves as an interface to separate the encoding phase from the compiler frontend. The final phase translates the intermediate program representation into target machine code.

The integrated tool set Eli is based on this model: Solutions of subtasks are produced by Eli's generating tools or taken from libraries embedded in Eli. For a deeper discussion of the specification methods and solution strategies offered by Eli we refer to [7,8]. An idea on how to use Eli can also be obtained from the complete specification of a compiler frontend and interpreter given in the Appendix.

The subtasks of "structuring" and "semantic analysis" are identical for both construction of compilers and interpreters. The compiler's tasks "data mapping"

and "action mapping" refer to the concepts of a source language machine. If we carefully design the representation of the machine concepts the mapping tasks also coincide with those of interpreter construction. Hence, it should be possible to generate both a compiler frontend and an interpreter from the same specification of these tasks.

The approach presented here is based on a few systematic strategies for the implementation of the machine and for the specification of the "action mapping" task. In the following we take examples from a compiler and interpreter specification for a small imperative language given in the Appendix.

3 Action mapping

The task of action mapping describes the effects of executable source language constructs in terms of source machine operations. The abstract program tree is mapped to a sequence of instructions. Each instruction identifies a machine operation to be applied to operands that refer to the data model of the machine.

Assume that our machine has a stack for expression evaluation. Then an expression that is a `Number` would be mapped to a push instruction

```
STMPushVal (Number.Sym)
```

Instruction sequences for non-leaf tree contexts are described by composition of the instruction sequences of their components, e. g. for a binary operation

```
<Instructions of Expression1>
<Instructions of Expression2>
 STMBinOpr (BinOpr.Fct)
```

The composition rules may be more complex, e. g. inserting instructions from other parts of the tree. Runtime control flow decisions are described by machine operations that place labels in the instruction sequence and branch to such labels. A conditional statement

```
if (Expression) Statement1 else Statement2
```

would be mapped to the following instruction sequence

```
<Instructions of Expression>
 STMBranchF (ifelselab)
<Instructions of Statement1>
 STMJump (ifendlab)
 AtLabel (ifelselab)
<Instructions of Statement2>
 AtLabel (ifendlab)
```

where `ifelselab` and `ifendlab` are labels generated for each application of that instruction sequence.

In Eli such sequencing is easily described using specifications based on attribute grammars: computations, in this case machine operation calls, are associated to contexts of the abstract program tree. The sequencing of such computations, i. e. the composition of the instruction sequences here, is specified by attribute dependencies. The instruction sequence for integral numbers would then read

```
RULE: Expression ::= Numb COMPUTE
  Expression.Eval =    STMPushVal (Numb.Sym)
        DEPENDS_ON Expression.Eval;
END;
```

The DEPENDS_ON clause describes the precondition of the computation, the attribute on the left of the = is a name for its postcondition. They both refer to a dependency chain threaded through the tree left-to-right depth-first, if not specified otherwise. (For explanation of the CHAIN construct see [6]). Here the pre- and postconditions insert the function call into the dependency chain Eval which links the execution of the machine operations throughout the tree. The same technique is used to translate the code sequence for the if statement into a specification of dependent computations:

```
RULE: Statement ::= 'if' '(' Expression ')' Statement 'else' Statement
COMPUTE
  Statement[2].Eval =  STMBranchF (.ifelselab)
       DEPENDS_ON Expression.Eval;
  Statement[3].Eval =
       ORDER (         STMJump (.ifendlab),
                       ATLabel (.ifelselab))
       DEPENDS_ON Statement[2].Eval;
  Statement[1].Eval =  ATLabel (.ifendlab)
       DEPENDS_ON Statement[3].Eval;
END;
```

The instructions of the Expression subtree are inserted at the beginning of the sequence by implicit CHAIN dependencies, while the ORDER construct describes sequencing within one computation.

4 Interpretation

The techniques described so far solve the compilation task: A language processor generated from such a specification produces the source machine code by executing calls of emit functions in the specified order. A conventional interpreter, like that for P-Code [1] would read the code into its instruction memory and execute a function call for each instruction while maintaining an instruction pointer.

In our approach, however, we integrate the interpreter into the translation phase. Instead of storing the instruction sequence in an instruction memory it is represented by the tree walk and the machine function calls. The calls are immediately executed, rather than emitted in an intermediate notation.

That kind of integration requires a new concept for execution of runtime control flow instructions. A jump instruction has to continue the translating tree walk in the state where the target label is generated. We achieve that by switching off any execution of machine function and continue the tree walk until the corresponding call that places the label is reached. For execution of a backward jump the whole translation process is repeated, switching the machine back to its execution mode when the label is reached. The iteration terminates when the end of the instruction sequence is reached in execution mode.

Such a repetition is easily achieved by our attribute grammar system LIGA [6], which allows for iterative evaluation of cyclically dependent computations. The iteration concept for cyclic attribute evaluation has been introduced in [11].

The root context of the abstract program tree has an instruction sequence that consists of a prologue and an epilogue with the code of the program embedded in between. Here the Eval dependency chain starts and ends:

```
RULE: Prog ::= Source COMPUTE
  Prog.Prologue =  STMAlloc (Prog.StoreSize)
      DEPENDS_ON Prog.GotLabel;

  CHAINSTART Source.Eval = Prog.Prologue;

  Prog.Epilogue = Label (Prog.Label)
      DEPENDS_ON Source.Eval;

  UNTIL Executing () ITERATE Prog.Prologue = Prog.Epilogue;
END;
```

The UNTIL ITERATE construct specifies that finally the condition Executing() holds. In order to establish that condition computations that depend on Prog.Prologue may be re-executed. For such re-execution a new "value" of that attribute is specified, here Prog.Epilogue describing the state where all machine operations for the program have been issued. This is a cyclic dependency causing iterative computations through the tree which has to be terminated by the UNTIL condition.

That iteration can also be understood as if the source machine executes several identical copies of the machine program concatenated, one additional copy for each executed backward jump.

5 Machine Implementation

The abstract source machine for this kind of interpretation is easily implemented. It differs from a conventional interpreter only in the technique of instruction representation: There is no instruction memory, no instruction pointer, and no reading of instructions. Instead the machine operates in one of two modes: executing or jumping. In the execute mode a function call performs the specified operation; in the jumping mode it returns immediately. For example the STMPushVal function would be implemented as follows:

```
void PushVal (int value)
{ if (!EXECUTE) return;
  Stack [StackPtr++] = value;
}
```

Execution of a jump instruction switches the machine to the jumping mode and stores the target label identification in a state variable TARGET

```
void STMJump (int label)
{ if (!EXECUTE) return;
  EXECUTE=0; TARGET = label;
}
```

Reaching the label instruction for the required target switches the machine back to the execute state:

```
void Label (int label)
{ if (TARGET == label) EXECUTE = 1;}
```

All machine operations are implemented according to this scheme. The example in the Appendix shows that a simple macro expansion can reduce the function description to the very semantics of the operations.

The execution mode concept of the machine can be extended to achieve other useful effects by introduction of further modes. Intermediate code can be produced if each function has a part that emits the instruction without switching the mode on jumps. A mode that produces trace information while executing could be introduced. The integration with the translation phase then allows to output source level information. In combination with Eli's facility to generate interactive processors debuggers can be constructed that stop the machine at certain breakpoints.

6 Efficiency

On the first glance it looks very inefficient to repeat the complete code generation for each execution of a backward jump. Hence, we need a closer look at its costs.

First of all we want to make sure, that no unnecessary computations are repeated during iteration. The evaluator generated by the LIGA system is a tree walking algorithm driven by visit-sequences [4,5]. The visit-sequences are computed such that they obey the specified dependencies between computations. The computations of the static semantics usually contribute only to the precondition of the iteration, and thus are not repeated in the cycle without further consideration in the specification. Hence the distinction between static and dynamic semantics is achieved automatically by dependency based visit-sequence computation. Our technique of attribute evaluator construction yields the increase of efficiency that other methods, e. g. based on denotational semantics, achieve by partial evaluation. In contrast to those approaches here no partially evaluated instances of the algorithm are constructed. Static and dynamic computations are kept together in one program where unnecessary computations are avoided by automatic dependency analysis.

Now let us consider the overhead resulting from the tree walking interpretation. The time for executing source machine function bodies is productive interpretation costs. Overhead is caused by the calls of these functions and by the tree walk operations. Subtree visits are implemented by calls of recursive procedures [5]. The evaluation chain usually needs one visit per node. Hence, for the four nodes and three machine instructions of an assignment `"a = 1"` within a statement sequence we pay the time of seven function calls overhead. That is about 1.4 microseconds out of 28 microseconds interpretation time for that assignment on a Sparc processor.

That is also the amount of overhead being paid when instructions are skipped in the jumping mode of the machine: one call for each tree node visit and one for each machine instruction that is skipped there. For example jumping across 100 assignments like `a = 1` would cost 700 calls, i. e. 1,4 milliseconds on a Sparc.

In the case of `goto` statements leading backwards or of procedure calls costs proportional to the program length are not avoidable in our approach. But the costs of backward jumps caused by source language loops can be drastically reduced. In this case we know that the generated target of the backward jump is within the same tree context where the jump is generated. Hence, we can apply the technique for iterative computation again for each loop. We add the specification of a local iteration to each loop context

```
UNTIL NOT (STMJumpingTo (.looplab)) ITERATE .Prelude = .Epilogue;
```

It has the same structure as the iteration in the root context. The UNTIL condition holds when the repeated execution of the loop is terminated, i. e. the machine is either in the execution mode or it is jumping out of the loop. This optimization reduces the costs for executing a loop iteration to be proportional to the length of the loop body rather than to the length of the whole program.

7 Conclusion

We have presented a systematic approach for generating interpreters and compilers from the same specification. It strictly follows the commonly accepted decomposition of language implementation tasks. As a consequence experience in compiler development and tools which support it, like Eli can be effectively reused for automized interpreter construction. Integration of both compiler and interpreter in one program allows to easily access compile time information at interpretation time. Hence, the approach can be extended to generate source level tracers and debuggers.

The technique of iterative evaluators generated from attribute grammar has proven to be suitable for interpreter generation. Well-known algorithms for attribute evaluator generation automatically separate static and dynamic semantics computations. The resulting interpreters are sufficiently efficient for practical use.

The example given in the appendix demonstrates that the approach is practicable. For ease of presentation we have omitted constructs like procedures and nested blocks from the example. Our experience with languages including such features are encouraging for applying the strategy to real-life languages.

8 References

1. Ammann, U., "On Code Generation in a PASCAL Compiler," *Software - Practice and Experience* 7 (1977), 391–423.
2. Gray, R. W., Heuring, V. P., Levi, S. P., Sloane, A. M. & Waite, W. M., "Eli: A Complete, Flexible Compiler Construction System," *Communications of the ACM* 35 (February 1992), 121–131.
3. Jones, N. D., Gomard, C. K. & Sestoft, P., *Partial Evaluation and Automatic Program Generation*, Prentice Hall, London, 1993.
4. Kastens, U., "Ordered Attributed Grammars," *Acta Informatica* 13 (1980), 229–256.
5. Kastens, U., "Implementation of Visit-Oriented Attribute Evaluators," in *Proceedings of the International Summer School on Attribute Grammars, Application and Systems*, Lecture Notes in Computer Science #545, Springer Verlag, New York–Heidelberg–Berlin, 1991, 114–139.
6. Kastens, U., "LIDO - Short Reference," University of Paderborn, FRG, Documentation of the LIGA System, 1992.
7. Kastens, U., "Executable Specifications for Language Implementation," in *Fifth International Symposium on Programming Language Implementation and Logic Programming, Tallinn, August 1993*, Lecture Notes in Computer Science #714, Springer Verlag, New York–Heidelberg–Berlin, 1993, 1–11.

8. Kastens, U., "Construction of Application Generators Using Eli," Universität-GH Paderborn, Reihe Informatik, Bericht Nr. 143, March 1994.
9. Knuth, D. E., "Semantics of Context-Free Languages," *Mathematical Systems Theory* 2 (June 1968), 127–146.
10. Lee, P., "Realistic Compiler Generation MIT Press," Cambridge, MA, 1989.
11. Meyer, M., "Visit-oriented evaluators for circular attribute grammars," Universität-GH Paderborn, Reihe Informatik, Bericht Nr. 69, April 1990.
12. Mosses, P. D., "SIS - Semantics Implementation System," Computer Science Department, Aarhus University, Technical Report DAIMI MD-30, 1979.
13. Mosses, P. D. & Watt, D., "The use of action semantics," in *Formal Description of Programming Concepts*, North-Holland, Amsterdam, 1987, 135–163.
14. Tennent, R. D., "The Denotational Semantics of Programming Languages," *Communications of the ACM* 19 (August 1976), 437–453.
15. Waite, W. M., Heuring, V. P. & Kastens, U., "Configuration Control in Compiler Construction," in *International Workshop on Software Version and Configuration Control '88*, Teubner, Stuttgart, 1988.

Appendix

This document demonstrates how to generate both a compiler frontend and an interpreter from one set of specifications using systematic techniques desribed in the main part of this paper.

The complete set of specifications used for Eli to generate a language processor are given here. The explanations emphasize the mapping of source language constructs to a source language machine and its execution. The specification of the analysis tasks is only briefly explained here. For more detailed information on the specification techniques used to solve those tasks, please refer to the documentation of the Eli system.

This example language defines programs that consist of a sequence of C-like statements and expressions using variables of integral type, e. g.

example[1] ≡
```
  {
y = 12;
x = y; fac = 1;
while (x > 1)
{
   fac = fac * x;
   x = x - 1;
}
output x;
output fac;
  }
```
This macro is attached to an output file.

The specified language processor maps such programs to the operations of a stack machine. It either produces the stack machine code or executes it immediately as requested when the processor is called.

Structuring Task

The program structure is defined by the following context-free grammar:

Inter.con[2] ≡
```
  {
Program:    Source .
Source:     Statements .

Statements: Statement Statements .
Statements: .

Statement:  DefLab ':' Statement .
Statement:  'goto' UseLab ';' .
Statement:  'if' '(' Expression ')' Statement 'else' Statement .
Statement:  'while' '(' Expression ')' Statement .
Statement:  UseVar '=' Expression ';' .
```

```
Statement:  '{' Statements '}' .
Statement:  ';' .

Statement:  'output' Expression ';' .
Statement:  'input' UseVar ';' .

Operand:    UseVar .
Operand:    Numb .

UseVar:     Ident .
DefLab:     Ident .
UseLab:     Ident .
    }
```

This macro is attached to an output file.

The last three productions distinguish different roles of identifiers. We have left out the part of the grammar that defines **Expressions**. A specification module taken from a library provides definitions of unary and binary operators on integral values in C-notation. The use of that library module is indicated by

Expr.specs[3] ≡

```
  {
$/Library/Expr.fw
    }
```

This macro is attached to an output file.

ELi derives the structure of the abstract program tree from the above concrete grammar, and generates operations to construct the tree.

Identifers integral numbers and comments are written as in C:

Inter.gla[4] ≡

```
  {
Ident:  C_IDENTIFIER
Numb:   C_INTEGER
        C_COMMENT
    }
```

This macro is attached to an output file.

Name Analysis

Variables and labels are named entities. Each occurrence of a name refers to the same object in the whole program. Variables are implicitly defined by using their name. Labels must have a unique definition that identifies a statement. There must be such a definition for each label used in a goto statement.

These language properties are specified by using specification modules of the name analysis library: the **Nest** module for consistent renaming of identifier occurrences, the **NoKeyMsg** module for checking identifiers to be defined, and the **Unique** module for checking definitions to be unique.

Scope.specs[5] ≡

```
  {
```

```
$/Tool/lib/Name/Nest.gnrc:inst
$/Tool/lib/Name/NoKeyMsg.gnrc:inst
$/Tool/lib/Name/Unique.gnrc:inst
    }
```

This macro is attached to an output file.

The modules provide certain name analysis concepts: The `Root` of a tree where name analysis is applied, `Ranges` that limit the scope of a definition, defining and applied occurences of identifiers, `IdDef`, `IdUse`.

Those concepts are related to our grammar symbols using the following spcification fragment, which is written in the attribute grammar specification language LIDO:

Scope.lido[6] ≡

```
  {
SYMBOL Program INHERITS RootNest END;
SYMBOL Source  INHERITS RangeNest, RangeUnique END;
SYMBOL UseVar  INHERITS IdDefNest, NoKeyMsg, IdentSym END;
SYMBOL DefLab  INHERITS IdDefNest, IdDefUnique, IdentSym END;
SYMBOL UseLab  INHERITS IdUseNest, NoKeyMsg, IdentSym END;

ATTR Sym: int;
SYMBOL IdentSym COMPUTE SYNT.Sym = CONSTITUENT Ident.Sym; END;
    }
```

This macro is attached to an output file.

Type Analysis

Our language has only one data type, i.e. integral numbers. Hence, there are no type rules to be checked. But there are two kinds of objects to be distinguished: Each name refers to an object of the kind variable or label, as required by the the context. We use the following encoding of those kinds:

Kind.head[7] ≡

```
  {
#define NoKind  0
#define VarKind 1
#define LabKind 2
    }
```

This macro is attached to an output file.

The kind property is associated to named objects by using the library module `KindSet`.

Kind.specs[8] ≡

```
  {
$/Tool/lib/Prop/KindSet.gnrc:inst
    }
```

This macro is attached to an output file.

The following LIDO fragment maps its concepts to grammar symbols and provides error messages on inconsistent uses.

Kind.lido[9] ≡

```
{
SYMBOL Program INHERITS RootKind END;
SYMBOL UseVar  INHERITS AddKind, GetKindSet COMPUTE
  THIS.Kind = VarKind;
  IF (InIS (LabKind, THIS.HasKindSet),
     message (ERROR, "variable also used as label", 0, COORDREF));
END;

SYMBOL LabName INHERITS AddKind, GetKindSet COMPUTE
  THIS.Kind = LabKind;
  IF (InIS (VarKind, THIS.HasKindSet),
     message (ERROR, "label also used as variable", 0, COORDREF));
END;

SYMBOL UseLab INHERITS LabName END;
SYMBOL DefLab INHERITS LabName END;
}
```

This macro is attached to an output file.

Data Mapping

Our source language machine has a trivial memory model: One memory entity for each integral variable is identified by numbers in the range of 0 to **MAXMEMORY**. Hence, we map each variable to its memory address by simply enumerating the variables in the program. That task is solved by a library module that counts objects. The module is instantiated with the generaic parameter **Var**, because another instance of that module is used below.

Alloc.specs[10] ≡

```
{
$/Library/ObjCnt.gnrc+instance=Var:inst
}
```

This macro is attached to an output file.

The variable addresses are made available at each use of a variable:

Alloc.lido[11] ≡

```
{
SYMBOL Program INHERITS VarRootObjCnt END;
SYMBOL UseVar  INHERITS VarObjCnt END;
}
```

This macro is attached to an output file.

Action Mapping

The machine implementation is driven by calls of the machine functions specified as computations in the node contexts of the program tree. Pre- and postconditions of those computations specify the desired sequencing of machine instructions.

Arbitrary unique numbers are used to identify positions in the instruction sequence of the source machine. The labels used in the source program are mapped to numbers by another instance of the `ObjCnt` module:

Labels.specs[12] ≡

```
{
$/Library/ObjCnt.gnrc+instance=Lab:inst
}
```

This macro is attached to an output file.

The label identifiactions are made available at defining and applied occurrences of labels:

Labels.lido[13] ≡

```
{
SYMBOL Program INHERITS LabRootObjCnt END;
SYMBOL DefLab  INHERITS LabObjCnt END;
SYMBOL UseLab  INHERITS LabObjCnt END;
}
```

This macro is attached to an output file.

Operations of the source language are mapped to sequences of calls of the functions that implement the machine operations. In the following specifications all function names that begin with `STM` refer to source machine functions. Their meaning is explained in the last section.

For each context of the abstract program tree an instruction sequence is specified. It describes the machine operations issued in that context and connects them with those issued in subtrees of the context.

That sequencing is specified using a dependency CHAIN

ChainEval.lido[14] ≡

```
{
CHAIN Eval:      VOID;
}
```

This macro is attached to an output file.

It describes a left-to-right depth-first depedency chain through the tree for all computations attached to it. In certain contexts that default order is overridden by explicitly specified dependencies.

The `Eval CHAIN` starts in the `Program` context. The `Prologue` which allocates storage for the variables of the program is the precondition for the operation sequence issued in the `Source` subtree:

ProgEval.lido[15] ≡

```
{
RULE: Program ::= Source COMPUTE
  Program.Prologue = STMAlloc (Program.VarMaxCnt)
      DEPENDS_ON Program.LabMaxCnt;

  CHAINSTART Source.Eval = Program.Prologue;
END;
}
```

This macro is attached to an output file.

A computation is attached to the `Eval CHAIN` by specifying its pre- and postcondition with respect to positions on the `CHAIN`. For example in

ExprUseVarEval.lido[16] ≡

```
{
RULE: Expression ::= UseVar COMPUTE
  Expression.Eval =  STMLoad ()
      DEPENDS_ON UseVar.Eval;
END;
}
```

This macro is attached to an output file.

The `STMLoad` operation pops an address from the stack and pushes the value found at that address in the memory. The operation is executed after the operation issued in the `UseVar` subtree where the address is pushed. Execution of the `STMLoad` operation establishes the postcondition for operations to be executed after this `Expression`.

UseVarEval.lido[17] ≡

```
{
SYMBOL UseVar COMPUTE
  THIS.Eval = STMPushVal (SYNT.VarObjCnt)
      DEPENDS_ON THIS.Eval;
END;
}
```

This macro is attached to an output file.

Here the precondition `THIS.Eval` denotes the `CHAIN` position before the `UseVar` leaf, and the postcondition `THIS.Eval` denotes the `CHAIN` position after the `UseVar` leaf. The `VarObjCnt` is the storage address of the variable as specified by the use of the `ObjCnt` module described above.

The remaining instruction sequences for `Expression` contexts are specified using the described patterns:

Expr.lido[18] ≡

```
{
RULE: Expression ::= Numb COMPUTE
  Expression.Eval =    STMPushVal (Numb.Sym)
        DEPENDS_ON Expression.Eval;
END;

RULE:  Expression ::=  Expression BinOpr Expression COMPUTE
  Expression[1].Eval = STMBinOpr (BinOpr.Fct)
        DEPENDS_ON Expression[3].Eval;
END;

RULE:  Expression ::=  UnOpr Expression COMPUTE
  Expression[1].Eval = STMUnOpr (UnOpr.Fct)
        DEPENDS_ON Expression[2].Eval;
END;
}
```

This macro is attached to an output file.

The value of the `Fct` attribute of an operator is a function that yields the result of the operation if applied to operands. It is passed as argument to the machine operation. The `Fct` attributes are specified in the expression library module mentioned above.

The instruction sequences for simple statements are obvious:

SimpleStmtEval.lido[19] ≡

```
{
RULE: Statement ::= UseVar '=' Expression ';' COMPUTE
  Statement.Eval =     STMStore ()
       DEPENDS_ON Expression.Eval;
END;

RULE: Statement ::= 'input' UseVar ';' COMPUTE
  Statement.Eval =
       ORDER (         STMRead (),
                       STMStore ())
       DEPENDS_ON UseVar.Eval;
END;

RULE: Statement ::= 'output' Expression ';' COMPUTE
  Statement.Eval =     STMWrite ()
       DEPENDS_ON Expression.Eval;
END;
}
```

This macro is attached to an output file.

The instruction sequences for statement labels and goto statements refer to the unique number of the label object which is specified by the use of the `ObjCnt` module as described above:

LabelEval.lido[20] ≡

```
{
SYMBOL DefLab COMPUTE
  THIS.Eval =          AtLabel (SYNT.LabObjCnt)
       DEPENDS_ON THIS.Eval;
END;

SYMBOL UseLab COMPUTE
  THIS.Eval =          STMJump (SYNT.LabObjCnt)
       DEPENDS_ON THIS.Eval;
END;
}
```

This macro is attached to an output file.

Source language control statements are translated using jump and conditional branch operations of the machine. Their targets are labels generated for each instance of the particular context. The **LabObjCnt** function provided by the **Lab** instance of the `ObjCnt` module is again used here to generate further unique label numbers.

The instruction sequence for the **if** statement uses two labels, one placed before the **else** statement, the other at the end of the **if** statement. Labels are placed in the instruction sequence using the **AtLabel** function of the machine.

IfEval.lido[21] ≡

```
    {
ATTR ifelselab, ifendlab: int;

RULE: Statement ::= 'if' '(' Expression ')' Statement 'else' Statement
COMPUTE
  .ifelselab = LabObjCnt ();
  .ifendlab = LabObjCnt ();

  Statement[2].Eval =   STMBranchF (.ifelselab)
       DEPENDS_ON Expression.Eval;
  Statement[3].Eval =
       ORDER (          STMJump (.ifendlab),
                        AtLabel (.ifelselab))
       DEPENDS_ON Statement[2].Eval;
  Statement[1].Eval =   AtLabel (.ifendlab)
       DEPENDS_ON Statement[3].Eval;
END;
    }
```

This macro is attached to an output file.

In the instruction sequence for the while statement one label is placed before the condition expression, the other at the end of the while statement.

WhileEval.lido[22] ≡

```
    {
ATTR looplab, loopendlab: int;

RULE: Statement ::= 'while' '(' Expression ')' Statement COMPUTE
  .looplab = LabObjCnt ();
  .loopendlab = LabObjCnt ();

  .Prologue = Statement[1].Eval;
  Expression.Eval =     AtLabel (.looplab)
       DEPENDS_ON .Prologue;
  Statement[2].Eval =   STMBranchF (.loopendlab)
       DEPENDS_ON Expression.Eval;
  .Epilogue =
       ORDER (          STMJump (.looplab),
                        AtLabel (.loopendlab))
       DEPENDS_ON Statement[2].Eval;
       End of the while instructions[24]
END;
    }
```

This macro is attached to an output file.

The end of the while instructions is described below.

Interpretation by Iterative Evaluation

The instruction sequences specified so far describe a compiler that translates the source program into machine code by calling the machine functions in the specified order. Each call can emit an instruction and its operands.

If the program is to be interpreted the machine function calls execute their operations rather than emit the instruction. Jumps turn the machine state from the executing mode into the jumping mode: Subsequent calls issued to the machine are not executed until a `AtLabel` call is reached such that its argument is the target of the jump.

Execution of backward jumps lead to the end of the code leaving the machine in the jumping mode. The target will be reached if the whole sequence of machine function calls is issued once more.

That iteration is specified using the `UNTIL ITERATE` construct in the `Program` context:

ProgIter.lido[23] ≡

```
{
RULE: Program ::= Source COMPUTE
  UNTIL Executing ()
    ITERATE Program.Prologue = Source.Eval;
END;
}
```

This macro is attached to an output file.

The `ITERATE` clause re-specifies the state attribute `Program.Prologue`, i.e. the begin of the code, to depend on `Source.Eval`, i.e. the end of the source program code `CHAIN`. Any computation that lies on this dependency cycle is re-executed until finally the `UNTIL` condition holds: The machine is then in its executing state at the end of the program.

In the same way we can improve the execution speed of the backward jumps issued for while loops:

End of the while instructions[24] ≡

```
{
Statement[1].Eval =
   UNTIL NOT (JumpingTo (.looplab))
     ITERATE .Prologue = .Epilogue;
   }
```

This macro is invoked in definition 22.

`Prologue` is specified to cyclically depend on `Epilogue`. The iteration establishes the following `UNTIL` condition: The machine is not jumping to the begin of the loop. The the while statement is executed as specified to be the postcondition of the `UNTIL ITERATE` computation.

Implementation of Machine Operations

The abstract source machine is implemented by two C modules: One provides the branching mechanism described above; it can be reused for any such source machine.

The other defines the set of functions and the data structures they operate on; it is specific for any variant of source machines.

The interface of the branching module supports three machine states executing (EXECUTE==1), jumping to label TARGET (EXECUTE==0), and emitting instructions (CODE==1).

Machine.h[25] ≡

```
{
extern int EXECUTE;
extern int TARGET;
extern int CODE;

#define Executing()       (EXECUTE || CODE)
#define JumpingTo(label) (!EXECUTE && TARGET==(label) && !CODE)
#define DoJump(label)     {EXECUTE=0; TARGET=(label);}

extern void AtLabel (int label);
}
```

This macro is attached to an output file.

Machine.head[26] ≡

```
{
#include "Machine.h"
}
```

This macro is attached to an output file.

The DoJump macro is exported to describe any kind of jump semantics in the function set of the specific machine.

The branching module is implemented by the following C code:

Machine.c[27] ≡

```
{
#include <stdio.h>
#include "Machine.h"

int EXECUTE = 1;
int TARGET = -1;

void AtLabel (int label)
{
  if (CODE) {printf ("%s\t%d\n", "Label", label); return; }
  if (TARGET==label) EXECUTE=1;
}
}
```

This macro is attached to an output file.

The emit mode of the machine can be switched on by giving the -c command line parameter on calling the processor. This facility is specified by

Machine.clp[28] ≡

```
{
CODE "-c" boolean;
```

```
    }
```

This macro is attached to an output file.

Our specific source machine operates on a memory for variables and a stack for expression evaluation:

Source Machine Data[29] ≡

```
  {
#define MAXMEMORY 1024
static int        Memory[MAXMEMORY];
static int        MemTop = 0;

#define MAXSTACK  64
static int        Stack[MAXSTACK];
static int        StackPtr = 0;
    }
```

This macro is invoked in definition 32.

The machine functions are implemented such that they emit the instruction text if the processor is compiling. If the processor is interpreting the operation is execute or skipped depending on the execution mode.

Each function is formed according to a fixed pattern provided by a cpp macro `McFct`. Its arguments are the function name, the signature of its arguments, and two statements, one for emitting the instruction, and one for executing it.

Function code macro[30] ≡

```
  {
#define McFct(name,args,emit,exec) \
        void name args { \
        if (CODE) {emit; return;} \
        if (EXECUTE) {exec;} \
        }
    }
```

This macro is invoked in definition 32.

The semantics of the functions of our stack machine should be clear from the execute statements in each second line of the following definitons:

Machine Function Definitions[31] ≡

```
  {
McFct (STMJump, (int label), printf ("Jump\t%d\n", label),
       DoJump(label))

McFct (STMBranchF, (int label), printf ("BranchF\t%d\n", label),
       {if (!(Stack[--StackPtr])) DoJump(label);})

McFct (STMAlloc, (int varno), printf ("Alloc\t%d\n", varno),
       {MemTop += varno;})

McFct (STMStore, (), printf ("Store\n"),
       {Memory[Stack[StackPtr-2]] = Stack[StackPtr-1];
        StackPtr -= 2;})
```

```
McFct (STMLoad, (), printf ("Load\n"),
        {Stack[StackPtr-1] = Memory[Stack[StackPtr-1]];})

McFct (STMPushVal, (int value), printf ("LoadVal\t%d\n", value),
        {Stack[StackPtr++] = value;})

McFct (STMBinOpr, (int (*f)(int, int)), printf ("BinOpr\n"),
        {Stack[StackPtr-2] =
                (*f) (Stack[StackPtr-2], Stack[StackPtr-1]);
        StackPtr--;})

McFct (STMUnOpr, (int (*f)(int)), printf ("UnOpr\n"),
        {Stack[StackPtr-1] = (*f) (Stack[StackPtr-1]);})

McFct (STMWrite, (), printf ("Write\n"),
        {printf ("%d\n", Stack[--StackPtr]);})

McFct (STMRead, (), printf ("Read\n"),
        {int v; scanf ("%d", &v); Stack[StackPtr++] = v;})
    }
```

This macro is invoked in definitions 32 and 33.

The C module is composed of the above fragments:

STM.c[32] ≡

```
    {
#include <stdio.h>
#include "Machine.h"
#include "STM.h"
```

Source Machine Data[29]
Function code macro[30]
Machine Function Definitions[31]

```
    }
```

This macro is attached to an output file.

The `McFct` is redefined to describe the module interface:

STM.h[33] ≡

```
    {
#define McFct(name,args,emit,exec) extern void name args;
```

Machine Function Definitions[31]

```
    }
```

This macro is attached to an output file.

STM.head[34] ≡

```
    {
#include "Machine.h"
#include "STM.h"
    }
```

This macro is attached to an output file.

Reihe I N F O R M A T I K

97. Kutylowski, M./Wanka, R.: Periodic sorting on two-dimensional meshes. Januar 1992

98. Wanke, E.: Bounded tree-width and LOGCFL. Februar 1992

99. Höfting, F./Wanke, E.: Minimum cost paths in periodic graphs. März 1992

100. Wanke, E.: The complexity of connectivity problems on context-free graph languages. Mai 1992

101. Lettmann, T./Schmitgen, S.: Auswahl- und Positionierungsprobleme beim Konfigurieren und ihre aussagenlogische Beschreibung. Juni 1992

102. Kastens, U./Waite, W.M.: Modularity and reusability in attribute grammars. Juli 1992

103. Kastens, U./Pfahler, P. (eds.): International Workshop on Compiler Construction CC'92. Extended Abstracts. Oktober 1992

104. Lengauer, T./Heistermann, J.: Hierarchical compaction and the solution of parameterized path problems. Oktober 1992

105. Wichmann, F./Pfahler, P.: Compilation for fine-grained parallelism: A code generator for the Intel i860. Oktober 1992

106. Kleine Büning, H./Karpinski, M./Flögel, A.: Resolution for quantified Boolean formulas. November 1992

107. Flögel, A./Kleine Büning, H./Lettmann, T.: On the restricted equivalence for subclasses of propositional logic. November 1992

110. Buro, M./Kleine Büning, H.: Report on a SAT competition. November 1992

111. Reski, T.: Parallele Simulation des Backpropagation Lernalgorithmus. November 1992

112. Klasing, R./Lüling, R./Monien, B.: Com pressing cube-connected cycles and butter fly networks. Februar 1993

113. Klasing, R./Monien, B./Peine, R./Stöhr E.A.: Broadcasting in butterfly and De Bruijn networks. Februar 1993

114. Lüling, R./Monien, B.: Load balancing for distributed branch & bound algorithms Februar 1993

115. Feldmann, R./Hromkovič, J./Madhavapedd S./Monien, B./Mysliwietz, P.: Optimal algorithms for dissemination of information in generalized communication modes. Februar 1993

116. Feldmann, R./Mysliwietz, P.: The shuffle exchange network has a Hamiltonian path. Februar 1993

117. Diekmann, R./Lüling, R./Simon, J.: Problem independent — Distributed simulated annealing and its applications. Februar 1993

118. Hromkovič, J./Procházka, J.: On the optimality of Kung's convolution algorithm in some classes of systolic algorithms. März 1993

119. Pfahler, P.: Karel the Robot. An exercise in implementing language processors with Eli. März 1993

120. Reinefeld, A./Marsland, T.A.: Enhanced iterative-deepening search. März 1993

121. Ladkin, P.B./Reinefeld, A.: A symbolic approach to interval constraint problems. März 1993

122. Funke, R./Lüling, R./Monien, B./Lücking, F./Blanke–Bohne, H.: An optimized reconfigurable architecture for transputer networks. März 1993

123. Kröger, B./Lüling, R./Monien, B./Vornberger, O.: An improved algorithm to detect communication deadlocks in distributed systems. März 1993

124. Monien, B./Feldmann, R./Klasing, R./Lüling, R.: Parallel architectures: Design and efficient use. März 1993

125. Menzel, K.: Report on distributed virtual reality. April 1993

126. Menzel, K./Ohlemeyer, M.: Walking-through animation in three-dimensional scenes on massively parallel systems. April 1993

127. Lürwer–Brüggemeier, K./Meyer auf der Heide, F.: Capabilities and complexity of computations with integer division. Juni 1993

128. Diekmann, R./Menzel, K./Stangenberg, F.: How to implement distributed algorithms efficiently. Juni 1993

129. Hromković, J./Klasing, R./Stöhr, E.A.: Dissemination of information in vertex-disjoint paths mode. Part 1: General bounds and gossiping in hypercube-like networks. August 1993

130. Hromković, J./Klasing, R./Stöhr, E.A./Wagener, H.: Dissemination of information in vertex-disjoint paths mode. Part 2: Gossiping in d-dimensional grids and planar graphs. September 1993

131. Pardubská, D.: On the power of communication structure for distributive generation of languages. September 1993

132. Reinefeld, A.: A minimax algorithm faster than alpha-beta. September 1993

133. Pardubská, D.: Communication complexity hierarchies for distributive generation of languages. September 1993

134. Karp, R.M./Luby, M./Meyer auf der Heide, F.: Efficient PRAM simulation on a distributed memory machine. September 1993

135. Kleine Büning, H./Lettmann, T.: Search space and average proof length of resolution. November 1993

136. Hromković, J./Klasing, R./Pardubská, D./Unger, W./Wagener, H.: The complexity of systolic dissemination of information in interconnection networks. Dezember 1993

137. Hromković, J./Kari, J./Kari, L./Pardubská, D.: Two lower bounds on distributive generation of languages. Januar 1994

138. Feldmann, R./Mysliwietz, P./Monien, B.: Game tree search on a massively parallel system. Januar 1994

139. Feldmann, R./Mysliwietz, P./Monien, B.: Experiments with a fully distributed chess program. Januar 1994

140. Nagel, C.: Selection of test points during high-level synthesis. Februar 1994

141. Buro, M.: The 1st International Paderborn Computer–Othello Tournament. Februar 1994

142. Dunker, U./Flögel, A./Kleine Büning, H./Lehmann, J./Lettmann, T.: ILFA – A project in experimental logic computation. März 1994

143. Kastens, U.: Construction of application generators using Eli. März 1994

144. Szepietowski, A.: Weak space complexity classes are not closed under complement. April 1994

145. Denejko, P./Diks, K./Pelc, A./Piotrów, M.: Reliable minimum finding comparator networks. Mai 1994

146. Kleine Büning, H. (Hrsg.): Workshop der GI–Fachgruppe Logik in der Informatik. Mai 1994

147. Wanka, R. (Hrsg.): 23. Workshop über Komplexitätstheorie, Datenstrukturen und effiziente Algorithmen. Mai 1994

148. Röttger, M./Schroeder, U.-P./Simon, J.: Virtual topology library for PARIX. Juni 1994

149. Röttger, M./Schroeder, U.-P./Unger, W.: Embedding 3-dimensional grids into optimal hypercubes. Juli 1994

150. Czumaj, A./Gąsieniec, L./Piotrów, M./Rytter, W.: Sequential and parallel approximation of shortest superstrings. August 1994

151. Kastens, U.: Generating interpreters fro compiler specifications. August 1994

Reihe FORSCHERGRUPPE

1. Höfting, F./Wanke, E.: Minimum cost paths in periodic graphs. März 1992

2. Wanke, E.: Paths and cycles in 3-dimensional conditional non-bounded dependence graphs. Dezember 1992

3. Meyer auf der Heide, F./Oesterdiekhoff, B./Wanka, R.: Strongly adaptive token distribution. Februar 1993

4. Monien, B./Lüling, R./Langhammer, F.: A realizable efficient parallel architecture. März 1993

5. Wachsmann, A./Wichmann, F.: OCCAM-light — A multiparadigm programming language for transputer networks. April 1993

6. Meyer auf der Heide, F.: Hashing strategies for simulating shared memory on distributed memory machines. Juni 1993

7. Kik, M./Kutylowski, M./Stachowiak, G.: Periodic constant depth sorting networks. Oktober 1993

8. Diekmann, R./Monien, B./Preis, R.: Using helpful sets to improve graph bisections. Mai 1994