Generic Compilation Schemes for Simple Programming Constructs[*]

A. Dold, F. W. von Henke, H. Pfeifer, H. Rueß

Fakultät für Informatik

Universität Ulm

D-89069 Ulm, Germany

{dold,vhenke,pfeifer,ruess}@informatik.uni-ulm.de

revised version of technical report
Ulmer Informatik-Berichte No. 96-12

January 27, 1999

In this paper we present a hierarchy of verified generic theories for compiling standard imperative language constructs using the specification and verification system PVS. The hierarchy consists of specifications for compiling assignments, control structures, and procedures into linearized assembly code. The specifications are generic in the sense that they abstract from concrete source and target languages; they specify an abstract compilation pattern which can be instantiated. Since each of these patterns can be formalized and verified separately, the verification task is broken into small manageable steps. A further modularization and reduction of complexity is achieved by splitting the compilation of control structures into three steps: control structures are first translated into a structure of blocks, then the blocks are linearized by introducing jump instructions, and finally, the procedures are linearized. Applicability of the generic theories to specific compilation processes is demonstrated by means of two simple examples.

[*]This research has been funded in part by the Deutsche Forschungsgemeinschaft (DFG) under project "*Verifix*"

Contents

1 Introduction

Verification of compiler correctness is a much-studied area. Many different approaches have been taken, usually with mechanized support to manage the complexity of specifications and proofs (e.g., [2–4,6,8–12,18–21,23]). Most of these studies address the verification of a specific compiler for a particular language; furthermore, specifications and proofs tend to be monolithic and lacking modular structure. As a consequence, it is difficult, if not impossible, to reuse parts of those specifications and proofs in similar verification tasks. On the other hand, both the compilation of standard constructs of imperative programming languages and the corresponding correctness proof usually follow a scheme that differs at most in small details among languages. For instance, the compilation of the assignment statement $x := e$ typically involves generating code for the evaluation of the expression e, followed by generating code for storing the computed value at the location denoted by x. The correctness of the latter process is independent of the structure and compilation of expressions, it depends only on the correctness of the code for evaluating and providing the expression's value. By separating concerns, abstracting from irrelevant details and concentrating on the essence of compilation steps, generic compilation patterns can be identified, formalized and verified. Ideally, if a sufficiently rich set of such generic theories for the compilation of different language constructs is available, a verified code generator for specific source and target languages can then be obtained by suitable composition and instantiation of existing pieces.

In this paper we present steps in the direction of developing such generic compiling theories. Specifically, we develop generic compilation patterns for elementary constructs of procedural languages, including

- simple statements, i.e. assignments,
- the standard control structures: sequencing, conditionals, loops, and
- (parameterless) procedures.

The theories form a kind of generic hierarchy in that the latter theory builds on the existence, but not the details, of the former; this actually results in a reduction of the overall verification effort.

The formal development has been carried out with the assistance of the specification and verification system PVS [15, 16]. PVS is particularly suitable for the task because it provides the necessary constructs for specifying parameterized theories, including constraining assumptions on parameters. The compilation theories presented in this paper are parameterized by source and target language and, as needed, by the compilation of constructs lower in the hierarchy; the assumptions on parameters are stated in such a manner that abstract compilation theorems can be derived. When a parameterized theory is instantiated, the PVS system takes care of generating the verification conditions required to demonstrate that the assumptions on parameters are satisfied. Then, for a specific source and target language a complete correctness proof can be established by combining the abstract compilation theorems.

The remainder of this paper is organized as follows: the following subsection summarizes work most closely related to our approach. Then a brief description of the PVS system is given. Section 2 presents the basic concepts needed in this paper. Section 3 gives an overview of the complete compilation structure and the hierarchy of PVS theories for compiling the imperative language constructs. In the following sections these theories are then described in greater detail. In Section 4 a generic specification of an operational semantics of linear code based on the single effects of the instruction set is presented. Section 5 deals with the compilation of simple statements, i.e. assignments. Section 6, 7, and 8 are concerned with the compilation of standard control structures. First, the translation into a basic block structure is outlined, and then we focus on the linearization phase and finally, procedures are linearized. To illustrate the application of the generic theories to specific compilation processes, compilation of a simple imperative language into code of a stack machine and a one-address machine is presented in section 9. Finally, Section 10 contains a short summary and an outlook. All theorems, lemmas and proof obligations have been completely proved. PVS theories and proof scripts may be obtained from the first author upon request.

Related Work

P. Curzon [6] verifies the compilation of a structured assembly language, Vista, into code for the VIPER microprocessor using the HOL system. Vista is a low-level language including arithmetic operators which correspond directly to those available on the target architecture. The specifications of the languages are generic only in the sense that they abstract from the specific word size, and the set of arithmetic and comparison operations available on the target machine.

P. Windley [22] uses a generic microprocessor specification in the context of microprocessor verification; he abstracts from the state and effects of individual instructions and provides a definition of an interpreter and various correctness predicates relating the different microprocessor levels. His generic interpreter is related to our generic specification for interpreting linear machine code as described in Section 4.

The modularization aspect of compiler verification has been considered by Müller-Olm [14], who deals with verifying the compilation process of an imperative real-time language into transputer code. Modularization is achieved by a stepwise derivation of increasingly abstract views of transputer behavior starting from a base model. The different levels then permit separate treatment of particular aspects. This approach, which can be adapted to other target architectures, is concerned with different abstraction levels for the target machine; this is in contrast to the approach presented here, which focuses on separating the compilation of different language constructs.

In [13], Müller-Olm considers the translation of control structures of a simple while language into linear machine code with relative jumps. Starting from a denotational semantics of while programs and an operational machine semantics, the machine semantics is characterized denotationally and then the equivalence of source and target semantics is established. Proofs are carried out without any mechanical support. Our proof of the linearization step presented in Section 7 is similar to these ideas.

A Brief Description of PVS

This section provides a brief overview of PVS. For more details consult [15, 16].

The PVS system combines an expressive specification language with an interactive proof checker that has a reasonable amount of theorem proving capabilities. The PVS specification language builds on classical typed higher-order logic with the usual base types, `bool`, `nat`, `rational`, `real`, among others, and the function type constructor `[A -> B]`. The type system of PVS is augmented with *dependent types* and *abstract data types*. A distinctive feature of the PVS specification language are *predicate subtypes*: the subtype {`x:A | P(x)`} consists of exactly those elements of type `A` satisfying predicate `P`. Predicate subtypes are used, for instance, for explicitly constraining the domains and ranges of operations in a specification and to define partial functions.

Predicates in PVS are elements of type `bool`, and `pred[A]` is a notational convenience for the function type `[A -> bool]`. Sets are identified with their characteristic predicates, and thus the expressions `pred[A]` and `set[A]` are interchangeable. For a predicate `P` of type `pred[A]`, the notation `(P)` is just an abbreviation for the predicate subtype {`x:A | P(x)`}.

In general, type-checking with predicate subtypes is undecidable; the type-checker generates proof obligations, so-called *type correctness conditions* (TCCs) in cases where type conflicts cannot immediately be resolved. A large number of TCCs are discharged by specialized proof strategies, and a PVS expression is not considered to be fully type-checked unless all generated TCCs have been proved.

Proofs in PVS are presented in a sequent calculus. The atomic commands of the PVS prover component include induction, quantifier instantiation, automatic conditional rewriting, simplification using arithmetic and equality decision procedures and type information, and propositional simplification using binary decision diagrams. The `SKOSIMP*` command, for example, repeatedly introduces constants of the form `x!i` for universal-strength quantifiers, and `ASSERT` combines rewriting with decision procedures.

Finally, PVS has an LCF-like strategy language for combining inference steps into more powerful proof strategies. The strategy `GRIND`, for example, combines rewriting with propositional simplification using BDDs and decision procedures. The most comprehensive strategies manage to generate proofs fully automatically.

2 Basic Concepts

In this section we summarize the basic concepts needed in this paper. The first two subsections are based on [1, 17], where a more comprehensive treatment can be found.

Fixed-Point Theory

Defining the semantics of loops requires fixed-point theory. In [1] we have developed a comprehensive formalization of domain and fixed-point theory in PVS, including formalizations of complete partial orders, notions related to monotonic and continuous functions, Knaster-Tarski fixed-point theorems for monotonic functions and Scott's fixed-point induction for admissible predicates and monotonic functions. We state the fixed-point induction theorems used in this paper.

A partial order `<=` over `D` is a *pre-cpo* over `D` if for every chain in `D` the least upper bound exists. If, in addition, the type `D` has a least element `bottom` then the pair `(<=, bottom)` is called a *complete partial order* (cpo).

1

```
% d: VAR CPO[D]

fp_induction_mono: THEOREM
  LET (<=, bottom) = d IN
    FORALL(f: Monotonic(<=, <=), P: Admissible(<=)):
      (P(bottom) AND (FORALL(x: D): P(x) IMPLIES P(f(x))))
        IMPLIES P(mu(<=)(f))

fp_induction_mono_le: LEMMA
  LET (<=, bottom) = d IN
   FORALL(f: Monotonic(<=, <=), P: Admissible(<=)):
    (P(bottom) AND (FORALL (x: D): P(x) AND x <= f(x) IMPLIES P(f(x))))
      IMPLIES P(mu(<=)(f))

park: LEMMA f(x) <= x IMPLIES mu(f) <= x
```

A special case of fixed-point induction is Park's lemma, which is useful for proving that something contains a least fixed point.

The definition of the least fixed-point operator `mu` makes use of the predicate subtype concept of PVS by restricting the operator to only those functions for which the least fixed point exists. Hence, typechecking an expression containing operator `mu`, PVS generates a corresponding TCC.

2

```
fixpoint?(f)(x): bool = (f(x) = x)

lfp?(<=)(f)(x) : bool = fixpoint?(f)(x)
                         & (FORALL y: fixpoint?(f)(y) IMPLIES x <= y)

lfp_exists?(<=)(f): bool = nonempty?(lfp?(<=)(f))

LFP(<=, f)    : TYPE = (lfp?(<=)(f))
LFP_Exists(<=): TYPE = (lfp_exists?(<=))

mu(<=)(f: LFP_Exists(<=)): LFP(<=, f) = choose(lfp?(<=)(f))

mu_is_fixpoint: LEMMA FORALL (f: LFP_Exists(<=)):
 f(mu(<=)(f)) = mu(<=)(f)

mu(d)(f: Monotonic(<=(d), <=(d))): LFP(<=(d), f) = choose(lfp?(<=(d))(f))
```

Monotonic functions mapping into a cpo always have least fixed-points - known as the *Knaster-Tarski* theorem.

```
KnasterTarski: THEOREM LET <= = <=(d) IN                                     3
 FORALL(f: Monotonic(<=, <=)): lfp_exists?(<=)(f)
```

State Transformers

Semantic definitions presented in this paper are expressed as state transformers which are modeled as relations. A relation $R \subseteq A \times B$ is represented as a function mapping elements of type A to a set of elements of type B. Partial functions can be defined as a subtype of relations by restricting the range to sets with at most one element. The following definitions are taken from the library of PVS theories for specifying the semantics of imperative language constructs [17].

```
% A,B : TYPE                                                                 4
Relation : TYPE = [A -> set[B]]
srel : TYPE = Relation

deterministic?(S:set[B]) : bool = empty?(S) OR singleton?(S)

PartialFunction : TYPE = [A -> (deterministic?)]
strans : TYPE = PartialFunction
```

A cpo over `srel` can be defined using the cpo function constructor `=>`, the predicate cpo `Pred[B]`, and the discrete cpo over type `A`: the partial ordering is denoted by `<=`, and the bottom element is called `abort`.

```
srel  : CPO[[A -> set[B]]] = (discrete[A] => Pred[B])                       5

<=    : preCPO[srel]     = (<=(srel))
abort : Bottom[srel](<=) = (bottom(srel))
```

State transformers for imperative language constructs can be defined easily using these definitions. In the sequel let `sigma` denote the type of states on which programs operate. Sequential composition of two state transformers `f` and `g`, denoted by `f ++ g` is defined by relational composition. Sequencing is monotonic in both arguments, and the composition of two deterministic state transformers is deterministic.

```
image(R:srel, S:set[sigma]) : set[sigma] =                                   6
 { y:sigma | EXISTS (s:(S)): member(y,R(s)) }

++(f, g) : srel = LAMBDA (s:sigma): image(g,f(s))
```

The semantics of a conditional is obtained by lifting the boolean IF-expression:[1]

```
IF(b:pred[sigma], f,g:srel) : srel =                                         7
 LAMBDA (s:sigma): IF b(s) THEN f(s) ELSE g(s) ENDIF
```

[1]Note that `IF` is overloaded here. In PVS, for expression `IF(b,f,g)` the convenient notation `IF b THEN f ELSE g ENDIF` can be used.

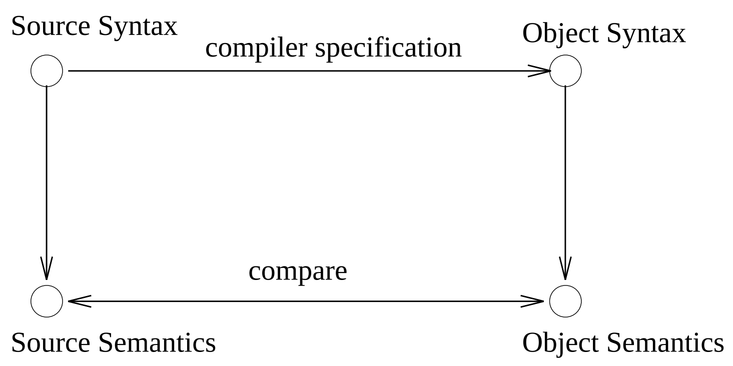

Figure 1: Compiler Specification Correctness

The semantics of loops is defined as usual as the least fixed point of a functional describing one iteration of the while-loop. The identity state transformer `skip` is modeled as a function mapping a state s to the singleton set {s}.

```
while(b:pred[sigma], f:srel) : srel =                                         8
  mu(srel)(LAMBDA (x:srel): IF b THEN f ++ x ELSE skip ENDIF)
```

In [17] we have proved that state transformers `++, IF` and `while` are deterministic if applied to deterministic state transformers.

In the correctness proof of the linearization step presented in Section 7, we will make use of the following *transfer* lemma:

```
F,G  : VAR {F1 : [srel -> srel] | monotonic?(<=,<=)(F1)}                      9
x,y  : VAR srel

transfer : LEMMA
 (FORALL x: (F(x) ++ y) = G(x ++ y))
   IMPLIES mu(srel)(F) ++ y = mu(srel)(G)
```

Notion of Correctness

The correctness of a compiler specification is generally understood as the commutativity of a kind of diagram as given in Fig. 1. Correctness is established by comparing the semantics of source programs and their compilations. As noticed, for example, by Chirica and Martin [5], this is only one aspect of compiler correctness. Another important aspect is the correctness of compiler implementation with respect to the specification. In this paper we concentrate on the correctness of compiler specifications.

There are many different possibilities to define the *compare* relation in Fig. 1. The definition used in this paper and in the Verifix project *preserves partial correctness of source programs*. It is claimed to be useful for verifying realistic compilation processes since it takes into account the finite resource limitations of real hardware. It allows the target machine program to fail if, for example, a memory or arithmetic overflow occurs. However,

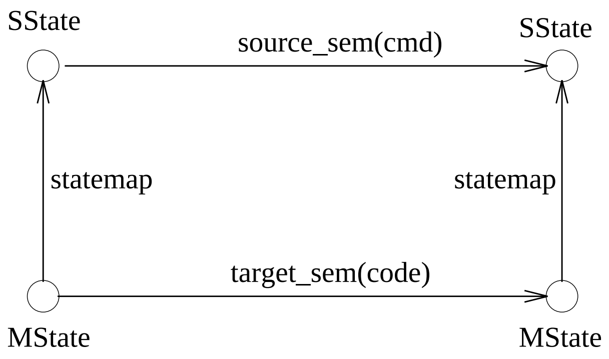
$$\begin{tikzcd}
\text{SState} \arrow[r, "\text{source\_sem(cmd)}"] & \text{SState} \\
\text{MState} \arrow[u, "\text{statemap}"] \arrow[r, "\text{target\_sem(code)}"] & \text{MState} \arrow[u, "\text{statemap}"']
\end{tikzcd}$$

Figure 2: Compiler Specification Correctness (refined)

whenever the execution of the compiled program produces a regular result, this result must correspond to the one produced by the source program. A more detailed discussion concerning this notion of correctness can be found in [7].

More precisely, let `Statement` denote the type of abstract source syntax (statements), `source_sem` its semantics given as a partial function on source states `SState`. Let further `Code` denote the type of target code, and `target_sem` its semantics given as a partial function on target states `MState`. In general, target states are different from source states, thus, we suppose that a mapping `statemap` from target states to source states $\boxed{10}$ is provided.

```
Statement        : TYPE,                                            10
SState           : TYPE,
source_sem       : [Statement -> PartialFunction[SState,SState]],
Code             : TYPE,
MState           : TYPE,
target_sem       : [Code -> PartialFunction[MState,MState]],
statemap         : [MState -> SState]
```

Then a generic notion of correctness in the sense of preservation of partial program correctness can be defined by predicate `pp_correctness` in $\boxed{11}$. It is illustrated in Fig. 2 which can be seen as an instance of the general correctness diagram in Fig. 1 since it gives a concrete definition of the `compare` relation. Informally, the predicate states that the target language semantics is contained in the source language semantics with respect to `statemap`.

```
% --- notion of correctness ---                                     11

pp_correctness(cmd:Statement)(code:Code) : bool =
 FORALL (start,final:MState):
  target_sem(code)(start)(final)
   IMPLIES
    source_sem(cmd)(statemap(start))(statemap(final))
```

3 Compilation Structure

Figure 3 illustrates the modular generic compilation of standard language constructs for imperative languages: expressions, simple statements (i.e. assignments), control structures, and procedures.

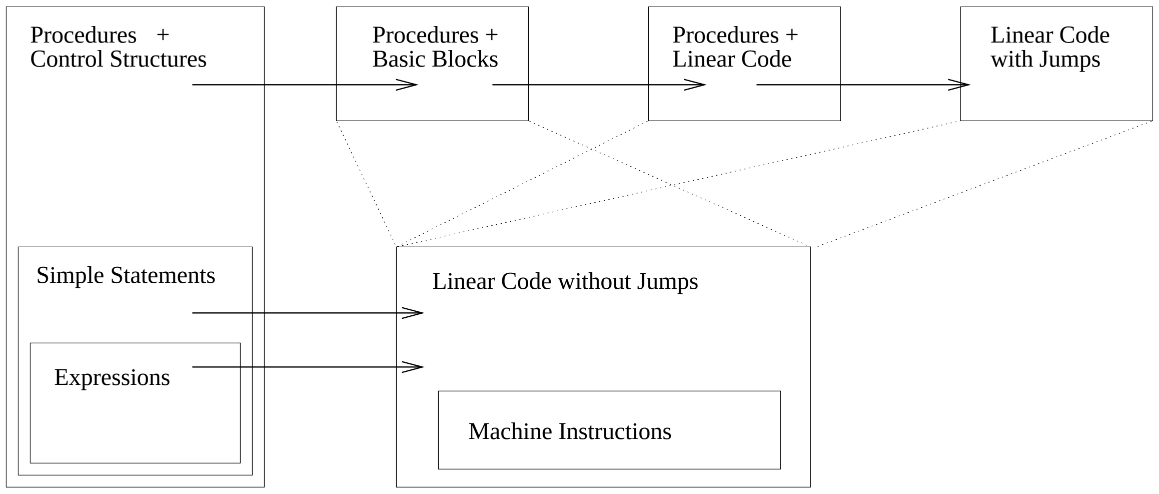

Figure 3: Generic Compilation Structure for Imperative Languages

On the lowest level in the hierarchy expressions are supposed to be compiled into linear machine code, i.e. code which does not contain branching instructions. The specification of the target machine is generic in the sense that it abstracts from the concrete instruction set and internal structure of the machine state. It is described in more details in the next section. In this paper we are not developing a *generic* theory for expression compilation; this will be considered in a future paper. However, in section 9 compilation of expressions into code of a stack machine and a one-address machine is described in details.

On the next higher level, compilation of simple statements is considered. Given a compilation function for expressions and assumptions about its correctness, the compilation of simple statements, i.e. assignments, need not take into account the structure of expressions or their compilation. All that is needed is a function providing access to the result of expression evaluation and an assumption that the accessed value is the correctly computed result. The theory specifying the compilation of simple statements is parameterized accordingly. This compilation step is described in more details in Section 5.

Similarly, the compilation of control structures (sequential composition, conditionals and loops) and procedures builds on compilation functions for expressions and simple statements; its correctness again depends on assumptions about the correctness of those compilation functions. For implementing control structures, branching instructions on the target architecture are required. It is convenient to split this compilation task into three steps: First, control structures are compiled into basic blocks. In a basic block graph the nodes consist of linear code sequences, and the edges represent the flow of control between these blocks. A basic block is assigned to each procedure. Basic blocks preserve the semantics of control structures. In a second step, basic blocks are then linearized by implementing the edges by the insertion of relative jumps. It has turned out that the introduction of such an intermediate language drastically simplifies the proof effort, in contrast to a direct translation of source statements into linear code with jumps. In addition, this step corresponds to a standard compilation phase in existing compilers. Linear machine code with relative conditional and unconditional jumps is assigned to each procedure. Finally, the procedure bodies and the main program are linearly ordered and jump tables are introduced.

4 Generic Interpreter for Linear Code

In this section a generic theory specifying the semantics of linear code (without jumps) for an "arbitrary" architecture is presented. In Figure 3 this PVS theory appears as the rightmost lower box.

```
simple_interpreter [Instr  : TYPE,                                    12
                    MState : TYPE+,
                    (IMPORTING relation[MState,MState])
                    effect : [Instr -> PartialFunction[MState,MState]]
                   ] : THEORY

BEGIN

  % -------- programs are lists of instructions

  Code : TYPE = list[Instr]

  c,l,k : VAR Code

  % -------  concatenation of code sequences
  ++(l,k) : Code = append(l,k)

  % -------- basic block interpreter

  interprete(c) : RECURSIVE srel =
   CASES c OF
    null      : skip,
    cons(i,r) : effect(i) ++ interprete(r)
   ENDCASES
    MEASURE length(c)

  interprete_deterministic : LEMMA
    FORALL (s:MState): deterministic?(interprete(c)(s))

  ...
END simple_interpreter
```

Theory `simple_interpreter` ($\boxed{12}$) abstracts from the specific instruction set, and the internal structure of the machine state (i.e. registers, memory, flags), and defines an operational semantics of linear code based on the single effects of the machine instructions given as a theory parameter. A deterministic state transformer `effect` is used for this purpose specifying the semantics of each machine instruction. The semantics of linear code sequences is then defined by a state transformer `interprete`. Lemma `interprete_deterministic` states that `interprete` is a deterministic state transformer. It can easily be proved using structural induction on the construction of lists. Note that the `++` operator, denoting sequential composition of state transformers, is overloaded here to define concatenation of code sequences.

5 Generic Compilation of Simple Statements

This section describes the generic specification and verification of simple statement compilation. A simple statement is given by an assignment of kind $x := e$. The generic compilation pattern consists of generating code for expression e and storing the value into a location for x. Therefore this compilation step can be specified with respect to the compilation of expressions. For specifying and proving correct this compilation step a set of parameters is required. They are grouped into parameters used for source and target language and the compilation process. Their meaning is given by a set of assumptions. In the following we describe the purpose of these parameters in more details.

Identifiers and expressions in assignments are represented by elements of type `Ident` and `Expr`, respectively. The concrete nature of expressions is irrelevant for this step. The semantics of expressions is assumed to be given by an evaluation function `eval` which takes an expression and an environment, a mapping from identifiers to values, as arguments and yields the value of the expression.

$\boxed{13}$
```
% === source ==========
Ident        : TYPE+,
Expr         : TYPE+,
SrcValue     : TYPE+,
eval         : [Expr -> [[Ident -> SrcValue] -> SrcValue]]
```

We suppose that simple statements are compiled into linear jump free code. Therefore, the semantics of target code can be defined using the generic interpreter for linear code; theory `simple_interpreter` $\boxed{12}$ is imported.

$\boxed{14}$
```
% === target ==========
Instr  : TYPE+,
MState : TYPE+,
effect : [Instr -> PartialFunction[MState,MState]]

IMPORTING simple_interpreter[Instr,MState,effect]
```

Additional parameters for the target language are required in order to specify access on target values and the memory. A parameter `output` is used abstracting from the specific value passing mechanism of expression values on the target architecture, i.e. the way how values on the target machine can be accessed. A stack machine, for example, accesses the value from the top of the stack, and in a one-address machine the value is accessed by reading the content of the accumulator, while a register machine reads the content of a specific register assigned by a register allocator. Another possibility is to store the expression value in memory and to access a specific memory cell. Here, type parameter T stands for the type of registers or memory addresses. For stack machines and accumulator machines this parameter is not required and can be instantiated with, for example, a unit type, see section 8 for examples. Values can generally be accessed only in those states in which `output` gives a defined value, as characterized by the predicate `outputdefd?`. For example, accessing a value in a stack machine requires the stack to be nonempty.

```
% --- access of target values ---                                    15

TarValue     : TYPE+,
T            : TYPE+,
outputdefd?  : [T -> pred[MState]],
output       : [t:T -> [ms:(outputdefd?(t))  -> TarValue]]
```

Parameter `Addr` abstracts from the type of memory addresses, and the target memory is given as a mapping from target addresses to target values. In addition, a function `STORE` is used for specifying the target code sequence for storing values provided at a specific location (of type T) at a specific memory address. Its meaning is specified by assumption `interprete_store`. Informally this assumption states that if the interpretation of the store code starting in a state `start` ends in a state `final`, then the memory is updated at the specific address with the value provided by `output`.

```
% === memory ============                                            16
Addr         : TYPE+,
STORE        : [T, Addr -> list[Instr]],
Mem          : [MState -> [Addr -> TarValue]]

interprete_store : ASSUMPTION
 FORALL (rn:T, a:Addr, start, final:MState):
   interprete(STORE(rn,a))(start)(final)
    IMPLIES
      outputdefd?(rn)(start)
        IMPLIES
          Mem(final) = Mem(start) WITH [(a) := output(rn)(start)]
```

Consider now the parameters used for specifying the compilation step. Compilation requires that source values are represented on the target architecture. Since realistic machines have restricted resources in general not all source language values are representable on the target architecture. For example, if the domain of source values is the set of integers, only a subset can be represented on a real target architecture. A predicate `representable?` is introduced for this purpose. We assume that a bijective function `valmap` is given mapping target values to representable source values.

Source language identifiers have to be mapped onto memory addresses. For this purpose, a function `idmap` is introduced. In addition, target states have to be related to source states (`statemap`)

```
% === compilation ===                                              17
representable? : pred[SrcValue],
valmap         : (bijective?[TarValue, (representable?)]),
idmap          : [Ident -> Addr],
statemap       : [MState -> SState]
```

The meaning of these parameters is specified by two assumptions:

```
statemap_and_memory : ASSUMPTION                                   18
 FORALL (ms1,ms2:MState, id,id1:Ident, v1:TarValue):
  Mem(ms2) = Mem(ms1) WITH [(idmap(id)) := v1] & id1 /= id
   IMPLIES statemap(ms2)(id1) = statemap(ms1)(id1)

symtab_and_memory : ASSUMPTION
 FORALL (ms:MState, id:Ident):
  valmap(Mem(ms)(idmap(id))) = statemap(ms)(id)
```

- Assumption `statemap_and_memory` expresses the fact that memory updates at variable addresses do not change the values of other variables with respect to `statemap`.
- Assumption `symtab_and_memory` states the relation between `idmap` and the state map: the contents of the memory address associated with an identifier corresponds to the value of the identifier with respect to `valmap`.

Finally, we assume that a compilation function `compileExpr` mapping expressions to code sequences is given. Additionally, this compilation function must provide the location on the target machine from which the expression's value is accessible. This location is fix for stack machines and accumulator machines, since the value is always accessed from the stack and accumulator, respectively. However, for other machines the value can be accessed from a specific register or memory cell.

```
% --- compiling function for expressions ---

compileExpr    : [Expr -> [(deterministic?[list[Instr]]), T]]
```

All what is assumed for this function is that it is correct in the sense of preservation of partial program correctness:

```
expression_compilation_correct : ASSUMPTION                        19
 FORALL (e:Expr):
  LET (c_set, resnr) = compileExpr(e) IN
   FORALL (c: (c_set)), (start,final:MState):
    interprete(c)(start)(final) IMPLIES
      outputdefd?(resnr)(final) AND
       valmap(output(resnr)(final)) = eval(e)(statemap(start)) AND
        statemap(final) = statemap(start)
```

Informally, this states that whenever the target machine stops in a state after interpreting the code `c` the value of expression `e` can be accessed using `output` with respect to `valmap` and statemap. In addition, expression evaluation on the target machine must not have any effect on the corresponding source states.

Based on these parameters the compilation of assignments can be specified. First, syntax and semantics of simple statements have to be defined. Syntax and semantics of simple statements are given by type SimpleStatement and deterministic state transformer ss_meaning, respectively. Semantics of an assignment is defined as usual: assign(x,e) updates the current state by assigning the value of e to identifier x.

```
% --- syntax of simple statements                                       20
SimpleStatement   : DATATYPE
 BEGIN
  assign(ass_var:Ident, ass_exp:Expr) : assign?
 END SimpleStatement

% --- semantics of simple statements
ss_meaning(cmd:SimpleStatement) : PartialFunction[SState,SState] =
 LAMBDA (ss:SState):
  CASES cmd OF
   assign(id,e)  : singleton(ss WITH [(id) := eval(e)(ss)])
  ENDCASES
```

Partial function compile_SimpleStmt then defines the compilation of simple statements. First, the expression is compiled using compileExpr then the value is stored at an address provided by idmap.

```
% --- compilation of simple statements ---                              21

compile_simpleStmt(cmd:SimpleStatement) : (deterministic?[Code]) =
 CASES cmd OF
  assign(id,e) : LET (c_set, rn) = compileExpr(e) IN
                  c_set ++ singleton(STORE(rn, idmap(id)))
 ENDCASES
```

For stating the correctness of this step (in the sense of partial program correctness) the generic notion of correctness presented in Section 2, $\boxed{11}$ is instantiated.

```
% === import generic notion of correctness ===
IMPORTING correct[SimpleStatement, SState, meaning, Code, MState, interprete, statemap]
```

Thus, one has to prove

```
simple_statement_comp_correct : THEOREM                                 22
 FORALL (cmd:SimpleStatement, c:Code):
  compile_simpleStmt(cmd)(c) IMPLIES pp_correctness(cmd)(c)
```

The proof of this theorem is by unfolding definitions, rewriting the assumptions, and applying propositional simplification.

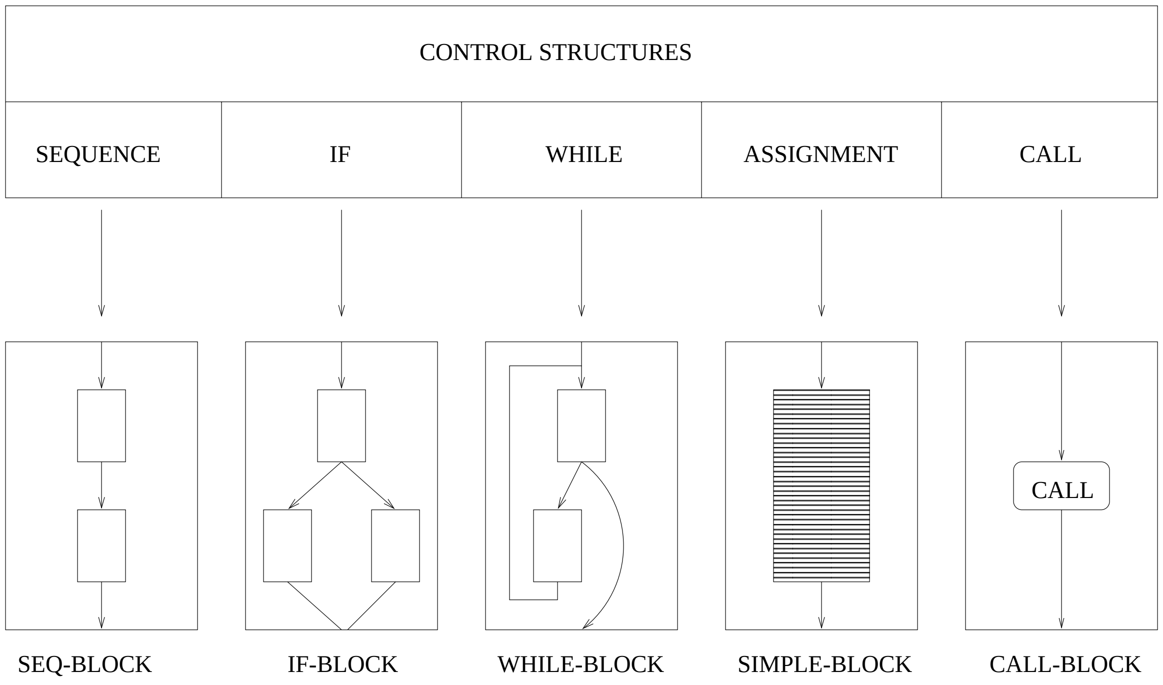

Figure 4: Compilation of Statements into Basic Block Graph

6 Compiling Control Structures into Basic Block Graphs

As stated in Section 3, the compilation of control structures is carried out in three steps: first, they are translated into a basic block structure, then that structure is linearized, and finally the procedures are implemented. In the next subsection we first present syntax and semantics of control structures (statements). Then we focus on basic blocks which have the same structure as statements but are defined on target language code. Finally, the compilation of control structures into basic blocks is outlined and proved correct.

Figure 4 illustrates this compilation step.

6.1 Control Structures

Consider the specification of source language control structures including simple statements, sequential composition, conditional, loop and (parameterless) procedures. The specification is parameterized with respect to the type and semantics of boolean expressions given by `BExp` and `eval` respectively, and the type and semantics of simple statements given by `SimpleStatement`, the type of procedure identifier `PId`, and the deterministic state transformer `ss_meaning`, respectively.

```
BExp                : TYPE,                                              23
SState              : TYPE+,
PId                 : TYPE+,
eval                : [BExp -> [SState -> bool]],
SimpleStatement     : TYPE,
ss_meaning          : [SimpleStatement -> PartialFunction[SState,SState]
```

The abstract syntax of control structures can then be defined in the obvious way using an abstract datatype `Statement`.

```
% --- syntax of control structures ---                                   24
Statement : DATATYPE
 BEGIN
  simple_stat(get_simple_stat:SimpleStatement)            : simple_stat?
  seq(first,second:Statement)                             : seq?
  itef(ifcnd: BExp, then_part, else_part: Statement)      : itef?
  while(whilecnd: BExp, while_body: Statement)            : while?
  call(p_name:PId)                                        : call?
 END Statement
```

Semantics of control structures is defined inductively in ($\boxed{25}$).

```
% --- semantics of control structures ---                                25
 meaning(c)(env) : RECURSIVE srel =
   CASES c OF
    simple_stat(si) : ss_meaning(si),
    seq(c1,c2)      : meaning(c1)(env) ++ meaning(c2)(env),
    itef(b,c1,c2)   : IF eval(b) THEN meaning(c1)(env) ELSE meaning(c2)(env) ENDIF,
    while(b,c1)     : while(eval(b), meaning(c1)(env)),
    call(i)         : env(i)
   ENDCASES
    MEASURE c BY <<
```

Semantics of the conditional and while statement are defined using state transformers `IF` and `while`, respectively (see Section 2, $\boxed{7}$, $\boxed{8}$). Procedure environments map procedure identifiers to state transformers. Environments build a cpo which is constructed using the cpo constructor `=>`.

```
%%% procedure environments, cpo, % Ident: TYPE+                          26
environment:TYPE = [Ident -> srel]
env_th: THEORY =  FP@exponent[Ident,srel]

env_cpo : CPO[[Ident -> srel]] = (discrete[Ident] => srel)
; <= : preCPO[environment] = (<=(env_cpo))
bottom:Bottom[environment](<=) = (bottom(env_cpo))
```

The semantics of procedure declaration is defined by instantiating a generic PVS theory `decl` for procedure declaration. The theory is parameterized with respect to the syntax and semantics of statements, and the type of identifiers:

27
```
%%% parameters of theory decl
statement:TYPE+,                        % --- syntax of statements
Ident:    TYPE+,                        % --- procedure identifier
sigma,tau:TYPE+,                        % --- types for state transformers
(IMPORTING env[Ident,sigma,tau])        % --- definition of environments
C: [statement -> [environment -> srel]]  % --- semantics of statements
```

Since the theory defines the environment for procedure declarations by a least fixed-point of a monotonic functional, the state transformer for statements is required to be monotonic with respect to the environment:

28
```
%%% assumption for the parameters
monotonic_prop : ASSUMPTION env1 <= env2 IMPLIES C(stat)(env1) <= C(stat)(env2)
```

The monotonic functional updating the environment is given by

29
```
decl_sem(d:[Ident -> statement]) : [environment -> environment] =
  LAMBDA env: LAMBDA i: C(d(i))(env)

decl_sem_monotonic: LEMMA monotonic?[environment,environment](<=,<=)(decl_sem(d))
```

Finally, the semantics of declarations is defined as the least fixed-point (over the environment cpo) of functional `decl_sem`: according to the Knaster-Tarski theorem this fixed-point exists.

30
```
D(d): environment = mu(env_cpo)(decl_sem(d))
```

We utilize the generic theory to define the semantics of procedure declarations for our source language. The meaning of a source program consisting of procedure declarations and a main program is defined by the meaning of the main program in the constructed procedure environment.

31
```
%%% semantics of procedure declarations
declsource: THEORY = decl[statement, PId, SState, SState, meaning]

source_program : TYPE = [# decls      : [PId -> statement],
                          main       : statement #]

%%% semantics of source programs
P(p:source_program): srel = meaning(main(p))(D(decls(p)))
```

6.2 Basic Blocks

For defining syntax and semantics of basic blocks the parameters in $\boxed{32}$ are used where the first four are used to specify the instruction set and its semantics; the generic interpreter theory `simple_interpreter` is imported ($\boxed{12}$). In addition, an access function for boolean values on the target machine is required. Such a boolean value is usually accessed by testing a specific flag which is set according to the result of the last executed operation. Here, we suppose that a boolean `output` function is given. Note, that the `output` function does only depend on the current machine state, i.e. the boolean value is accessed from a constant target location (specific flag, accumulator, top of stack etc.) Again, a predicate (`outputdefd?`) is introduced denoting the set of states in which an access is possible. Additionally, a function `read` specifies the state transition carried out when accessing a value using `output`. For example, in a stack machine a value is accessed by reading the stack's top element followed by a pop operation. In a register machine, parameter `read` would be instantiated to the identity function on states.

```
%%% parameterization of basic block graphs                                32
Instr         : TYPE,
MState        : TYPE+,
PId           : TYPE+,
effect        : [Instr -> PartialFunction[MState,MState]],
outputdefd?   : pred[MState],
output        : [(outputdefd?) -> bool],
read          : [(outputdefd?) -> MState]
```

A basic block graph is defined in 33 using an abstract datatype with five constructors each representing one specific control structure. A basic block graph is assigned to each procedure.

```
% --- syntax of basic blocks ---                                          33
bb_graph : DATATYPE
 BEGIN
  simple_block(code_seq  : Code)           : simple_block?
  seq_block(fst,scd  : bb_graph)           : seq_block?
  if_block(if_cnd, thn, els : bb_graph)    : if_block?
  while_block(while_cnd, body : bb_graph)  : while_block?
  call_block(pid:PId)                      : call_block?
 END bb_graph
```

In particular,

- a simple block consists of a linear code sequence.
- a sequential block consists of two subblocks.
- a conditional block consists of three subblocks, one block representing the condition, and two blocks denoting the true and false block, respectively.
- a while block consists of a condition block and a block denoting the body of the loop,
- a call block consists of the single call statement.

The "nodes" of the graph are given by the different block constructors, whereas the "edges" are given by the semantics. It is defined by a (deterministic) state transformer `bb_ip` defined on machine states.

```
%%% semantics of basic blocks %%%                                                 34
bb_ip(g)(env): RECURSIVE srel =
  CASES g OF
    simple_block(p)    : interprete(p),
    seq_block(b1,b2)   : bb_ip(b1)(env) ++ bb_ip(b2)(env),
    if_block(c,t,e)    : IF bb_ip(c)(env) THEN bb_ip(t)(env) ELSE bb_ip(e)(env) ENDIF,
    while_block(c,bd) : while(bb_ip(c)(env), bb_ip(bd)(env)),
    call_block(i)      : env(i)
  ENDCASES
   MEASURE g BY <<
```

More specifically, the semantics of simple blocks is defined using `interprete` which defines the semantics of linear code sequences as given in theory `simple_interpreter`. Sequential blocks are interpreted by relational composition (`++`) of the semantics of the two subblocks. The semantics of the if-block reflects the semantics of a conditional statement. It is given by relational composition of the semantics of the condition subblock and a branch state transformer `fork` which branches to the first or second subblock depending whether the output is true or false in a specific state in which a value can be accessed. If the output is not defined in this state function `fork` returns the empty set.

```
% --- state transformer fork ---                                                  35
fork(f,g) : srel =
   ( LAMBDA ms: IF outputdefd?(ms) THEN
                 IF output(ms) THEN f(read(ms))
                 ELSE g(read(ms))
                 ENDIF
                ELSE emptyset[sigma]
                ENDIF )

IF(st1,st2,st3) : srel = st1 ++ fork(st2,st3)
```

Semantics of the while-block is given by the least fixed point `mu` of a functional `while` which has exactly the same structure as the corresponding source language state transformer.

```
% --- state transformer while ---                                                 36

while(st1,st2): srel =
 mu(srel)(LAMBDA (h:srel): IF st1 THEN st2 ++ h ELSE skip ENDIF)
```

It has to be proved that state transformer `bb_ip` is deterministic. This has to be established for each case in the definition of `bb_ip`. For the first case this is trivial since we already have proved that `interprete` is deterministic. For the second case a lemma from the library can be utilized which states that deterministic state transformers are closed under composition. Deterministic state transformers are also closed under `IF` since they are closed under compositionality. For proving closure under `while` fixed-point induction for monotonic functions is required. The proof follows exactly the one given in [17] for `while_strans_closed`.

For defining the semantics of procedure declarations, the generic theory `decl` is instantiated. A basic block program consists of procedure declarations mapping identifiers to basic block graphs, and a main basic block graph.

```
bb_program: TYPE = [# pd: [PId -> bb_graph], main_block: bb_graph #]                37

decltarget: THEORY = decl[bb_graph, PId, MState, MState, bb_ip]

BS(bp:bb_program): srel = bb_ip(main_block(bp))(D(pd(bp)))
```

The following useful lemma states the characteristic behavior of the while block which corresponds to the behavior of the while statement.

```
bb_ip_while_block_unfold : LEMMA                                                    38
 bb_ip(while_block(w_cnd, w_body))(env) =
   IF bb_ip(w_cnd)(env)
    THEN bb_ip(w_body)(env) ++ bb_ip(while_block(w_cnd, w_body))(env)
    ELSE skip
   ENDIF
```

6.3 Compilation

Compilation of (abstract) source programs into basic block programs can be specified with respect to expression compilation and simple statement compilation. More specifically, the PVS theory specifying this compilation step abstracts from

- the syntax and semantics of (boolean) expressions,
- the syntax and semantics of simple statements,
- the instruction set and internal structure of the machine state,
- the value access function `output` and the corresponding access state transition `read`,
- the state map (`statemap`),
- the compilation of both (boolean) expressions `compileBExpr` and simple statements given as deterministic state transformers.

The parameters must satisfy the following assumptions:

- Both boolean expression compilation and simple statement compilation must be correct in the sense of preservation of partial correctness.
- The read function must have no effect on the corresponding source states with respect to `statemap`.

Partial function `compileStmt`, sketched in $\boxed{39}$ defines the compilation of control structures into the block structure. Not surprisingly, each control structure is compiled into the corresponding basic block. For example, a conditional `itef(b,c1,c2)` is compiled into a `if_block` where the boolean expression `b` is compiled into a simple block consisting of the code sequence which is the result of applying `compileBExpr` to `b`. A call statement is translated into a call block.

```
% --- compilation of control structures into basic blocks ---            39
compileStmt(cmd:Statement) : RECURSIVE (deterministic?[bb_graph]) =
   CASES cmd OF
    ....
    itef(b,c1,c2) :
      LET g1 = compileBExpr(b) IN
       IF empty?[Code](g1) THEN emptyset[bb_graph]
       ELSE
        LET g2 = compileStmt(c1) IN
         IF empty?[bb_graph](g2) THEN emptyset[bb_graph]
         ELSE LET g3 = compileStmt(c2) IN
          IF empty?[bb_graph](g3) THEN emptyset[bb_graph]
          ELSE
           singleton[bb_graph](if_block(simple_block(choose(g1)),
                                        choose(g2),choose(g3)))
          ENDIF
         ENDIF
        ENDIF,

    ...
    ENDCASES
     MEASURE cmd BY <<
```

Correctness of this compilation step is stated by

```
correctness: THEOREM                                                    40
 compile_defined?(p) AND
  BS(compile(p))(start)(final)
   IMPLIES P(p)(statemap(start))(statemap(final))
```

Whenever the compilation of a program is defined (predicate `compile_defined?`), and the meaning of the compiled program is defined in some final state, then also the semantics of the source program is defined in the corresponding source states (preservation of partial correctness).

In order to prove this conjecture, an auxiliary property is established stating correctness of statement compilation:

```
%%% auxiliary property for correctness proof                            41
cr1(p)(rho) : bool =
 FORALL cmd,g:
  FORALL start,final:
   compileStmt(cmd)(g) AND
     bb_ip(g)(rho)(start)(final)
       IMPLIES
        meaning(cmd)(D(decls(p)))(statemap(start))(statemap(final))

%%% auxiliary property for cr1, same as cr1 but constant cmd
  cr_aux(p)(rho)(cmd): bool =
   FORALL g:
    FORALL start,final:
     compileStmt(cmd)(g) AND
      bb_ip(g)(rho)(start)(final)
       IMPLIES
        meaning(cmd)(D(decls(p)))(statemap(start))(statemap(final))
```

Then the main goal to prove is the fact that for all (non-empty) compilations of procedures, property `cr1` holds in the procedure environment generated for the compiled procedure declarations.

```
%%% compilation of each procedure body is defined                          42
procbodies_defined?(p) : bool = FORALL i: not(empty?(compileStmt(decls(p)(i))))

%%% corresponding total function
compile_decls(p:(procbodies_defined?)) : [PId -> bb_graph] =
  LAMBDA i: choose(compileStmt(decls(p)(i)))

%%% main conjecture
cc: THEOREM procbodies_defined?(p) IMPLIES cr1(p)(D(compile_decls(p)))
```

The main conjecture `cc` is by fixed-point induction over the procedure environment. The base case for the 'bottom' environment is trivial and automatically proved by `grind`. Admissibility is by characterizing the least upper bound of a chain `C` of environments as the function mapping an identifier and a state to the union of all images of the functions in the chain `C`.

The induction step of this fixed-point induction is by structural induction on the structure of statements. To modularize the proof, for each step, separate compilation theorems have been introduced.

```
%%% compilation theorems for statements %%%                                43
simple_c      : VAR SimpleStatement
cmd,cmd1,cmd2 : VAR Statement
g,g1,g2       : VAR bb_graph
b             : VAR BExp
c             : VAR Code
rho           : VAR environment[PId, MState, MState]
p             : VAR source_program

simple_correct : THEOREM cr_aux(p)(rho)(simple_stat(simple_c))

sq_correct : THEOREM
   cr_aux(p)(rho)(cmd1) & cr_aux(p)(rho)(cmd2)
     IMPLIES cr_aux(p)(rho)(seq(cmd1,cmd2))

if_correct : THEOREM
   cr_aux(p)(rho)(cmd1) & cr_aux(p)(rho)(cmd2)
     IMPLIES cr_aux(p)(rho)(itef(b,cmd1,cmd2))

wh_correct : THEOREM
   cr_aux(p)(rho)(cmd) IMPLIES cr_aux(p)(rho)(while(b,cmd))

call_correct: THEOREM
  cr1(p)(rho) AND procbodies_defined?(p)
    IMPLIES cr_aux(p)(decl_sem(compile_decls(p))(rho))(call(i))
```

The proof of the first theorem is by expanding definitions and using the assumption about the correctness of simple statement compilation. The second theorem which states the correctness of sequential composition compilation is straightforward. The proof of the third theorem uses the assumption stating the correctness of expression compilation and the assumption about `read`. Here, the most interesting theorem is the compilation theorem for the while statement. It is proved using park's lemma for monotonic functions. Finally, the call lemma requires some characteristic properties about fixed-points.

7 Linearization of Basic Block Graphs

So far, statements have been compiled into basic blocks where the structure is preserved and a basic block is assigned to each procedure. The next step is to implement the control structure, i.e. to linearly order the subblocks of a basic block by introducing relative jumps. In the next subsection the (abstract) linear target code is extended by relative jump instructions and subroutine calls, and an operational semantics is defined. Then linearization of blocks is specified and proved correct.

7.1 Linear Target Code with Jumps

A new datatype `MInstr` ($\boxed{44}$) is defined which extends a simple linear instruction sequence (`lin_code(p)`) as defined in theory `simple_interpreter` by unconditional (`jmp`), conditional jumps (`jmc`), and call of subroutines, i.e. machine instructions are built from linear code sequences and jumps.

$\boxed{44}$
```
% --- code sequences with jumps ---
MInstr : DATATYPE
 BEGIN
  lin_code(get_ins : Code)  : lc?
  jmp(jp_adr:int)           : jmp?
  jmc(jc_adr:int)           : jmc?
  jsr(jsr_adr:PId)          : jsr?
 END MInstr
```

A linear machine program is assigned to each procedure identifier:

```
linear_code : TYPE+ = list[MInstr]

machine_program: TYPE = [# sdecls:[PId -> linear_code], main:linear_code #]
```

The semantics of linear code with jumps is based on the semantics of simple linear code (without jumps). Since jumps and jump subroutines have been introduced the abstract machine state given as an uninterpreted type `MState` has to be extended. An additional abstract stack structure is introduced to model subroutine calls and returns. Each stack entry consists of a tuple of linear code (denoting the body of the current procedure), and a (local) program counter which points to the machine instruction within the current procedure body. The extended machine state is called a *configuration*.

```
%%% machine configurations                                              45
Conf : TYPE = [MState, Stack[[linear_code, int]]]
```

An operational semantics for this machine is based on configuration transformations. First, a one-step configuration transformation is defined by the deterministic state transformer **eff** in $\boxed{46}$ specifying the effects of each instruction of type `MInstr`:

```
eff(mp)(c1,c2) : bool =                                                 46
 LET (ms1, s1) = c1, (ms2, s2) = c2 IN
   IF empty?(s1) THEN false
    ELSE
     LET t = top(s1), code = proj_1(t), n = proj_2(t) IN
   IF (n > length(code) OR n < 1) THEN (s2 = pop(s1) AND ms2 = ms1)
    ELSE
     CASES nth(code, n - 1) OF
      lin_code(p)  :  interprete(p)(ms1)(ms2) & s2 = push((code, n + 1), pop(s1)),
      jmp(i)       :  ms2 = ms1 & s2 = push((code, n + i), pop(s1)),
      jmc(i)       :  IF outputdefd?(ms1) THEN
                       IF output(ms1) THEN
                         ms2 = read(ms1) & s2 = push((code, n + 1), pop(s1))
                       ELSE
                         ms2 = read(ms1) & s2 = push((code, n + i), pop(s1))
                       ENDIF
                       ELSE false
                      ENDIF,
      jsr(i)       :  ms2 = ms1 &
                      s2 = push((sdecls(mp)(i), 1), push((code, n+1), pop(s1)))
     ENDCASES
   ENDIF
  ENDIF
```

Informally, the instructions have the following effects:

- there is no configuration transition if the current stack is empty
- if the program pointer points outside the current procedure body then the current procedure body is popped from the stack (return)
- `lin_code(p)` changes the current state by interpreting the linear code `p` using the function `interprete` and increments the program counter,
- `jmp(i)` updates the current program counter
- `jmc(i)` changes the state using function `read` and updates the program counter according to the output value provided that such a value can be accessed
- `jsr(i)` increments the current program pointer (return address), pushes the body of the called procedure onto the stack, and sets the pointer to the first instruction of the body.

The operational semantics of a linear code program is then given by repeatedly applying the above transition from a start configuration. This process may end in a final configuration where the stack is empty (the program terminates). In PVS, this semantics can be represented by a relation `M` in $\boxed{47}$ where `RTC(eff(mp))` is the reflexive and transitive closure of `eff(mp)`. Note that typechecking an inductively defined relation such as `RTC` generates a corresponding induction principle. In the following, let `RTC_induction` denote the induction principle for `RTC`.

```
M(mp)(s1:Stack)(ms1:MState)(ms2:MState) : bool =                               47
 RTC(eff(mp))((ms1, s1), ((ms2, empty)))
```

Since relation `eff` is deterministic, it can be proved using `RTC_induction`, that `M` is also deterministic and hence is a partial function.

Based on the closure of `eff` the semantics of a complete machine program is defined by a deterministic state transformer `Ip` on `MState` (the interpreter). The program `mp` is "executed" from start (i.e. main is pushed onto the stack, and the program counter is set to 1). Note that `Ip` is a state transformer on `MState` and does not depend on the program counter.

```
%%% interpreter for machine programs                                           48
Ip(mp) : strans = M(mp)(push((main(mp), 1), empty))
```

We state some simple consequences (laws) of this semantic definition which are convenient when proving the correctness of the linearization step.

- The machine behaves like skip when started with an empty stack:

```
law_identity  : COROLLARY  M(mp)(empty) = skip                                49
```

- If the current instruction is `lin_code(p)` then the total behavior can be described by relationally composing the interpretation of `p` using `interprete` with the computation starting at the instruction directly following `p`.

```
law_linear_code : COROLLARY                                                    50
 (n >= 1) & (length(u) >= n) & nth(u, n - 1) = lin_code(p)
   IMPLIES
     M(mp)(push((u, n), a)) = (interprete(p) ++ M(mp)(push((u, n + 1), a)))

law_linear_code1: LEMMA
 M(mp)(push((u ++ (: lin_code(p) :) ++ v, 1 + length(u)), a)) =
  interprete(p) ++ M(mp)(push((u ++ (: lin_code(p) :) ++ v, 2 + length(u)), a))
```

- Analogously there are laws for unconditional jumps

```
law_jmp : COROLLARY                                                          51
  (n >= 1) & (length(u) >= n) & nth(u, n - 1) = jmp(i)
    IMPLIES  M(mp)(push((u, n), a)) = M(mp)(push((u, n + i), a))

law_jmp1 : COROLLARY
   M(mp)(push((u ++ (: jmp(i) :) ++ v, 1 + length(u)), a)) =
   M(mp)(push((u ++ (: jmp(i) :) ++ v, 1 + length(u) + i), a))
```

- conditional jumps using semantic function `fork`, (see $\boxed{35}$)

```
law_jmc : COROLLARY                                                          52
      (n >= 1) & (length(u) >= n) & nth(u, n - 1) = jmc(i)
        IMPLIES
          M(mp)(push((u, n), a)) =
           fork(M(mp)(push((u, n + 1), a)), M(mp)(push((u, n + i), a)))
law_jmc1 : COROLLARY
  M(mp)(push((u ++ (: jmc(i) :) ++ v, 1 + length(u)), a)) =
     fork(M(mp)(push((u ++ (: jmc(i) :) ++ v, 2 + length(u)), a)),
          M(mp)(push((u ++ (: jmc(i) :) ++ v, 1 + length(u) + i), a)))
```

- jump subroutine, and

```
law_jsr : COROLLARY                                                          53
    (n >= 1) & (length(u) >= n) & nth(u, n - 1) = jsr(k)
     IMPLIES
      M(mp)(push((u, n), a)) = M(mp)(push((sdecls(mp)(k), 1), push((u, n + 1), a)))

law_jsr1: COROLLARY
  M(mp)(push((u ++ (: jsr(k) :) ++ v, 1 + length(u)), a)) =
  M(mp)(push((sdecls(mp)(k), 1), push((u ++ (: jsr(k) :) ++ v, 2 + length(u)), a)))
```

- return from a subroutine (note that there is no explicit return instruction)

```
return_law: COROLLAR (n < 1) OR (n > length(u))                              54
                      IMPLIES M(mp)(push((u, n), a)) = M(mp)(a)
```

7.2 Compilation

Consider now the compilation process from basic block graphs into linear code with jumps as illustrated by Fig. 5 and specified by function `lin` in $\boxed{55}$.

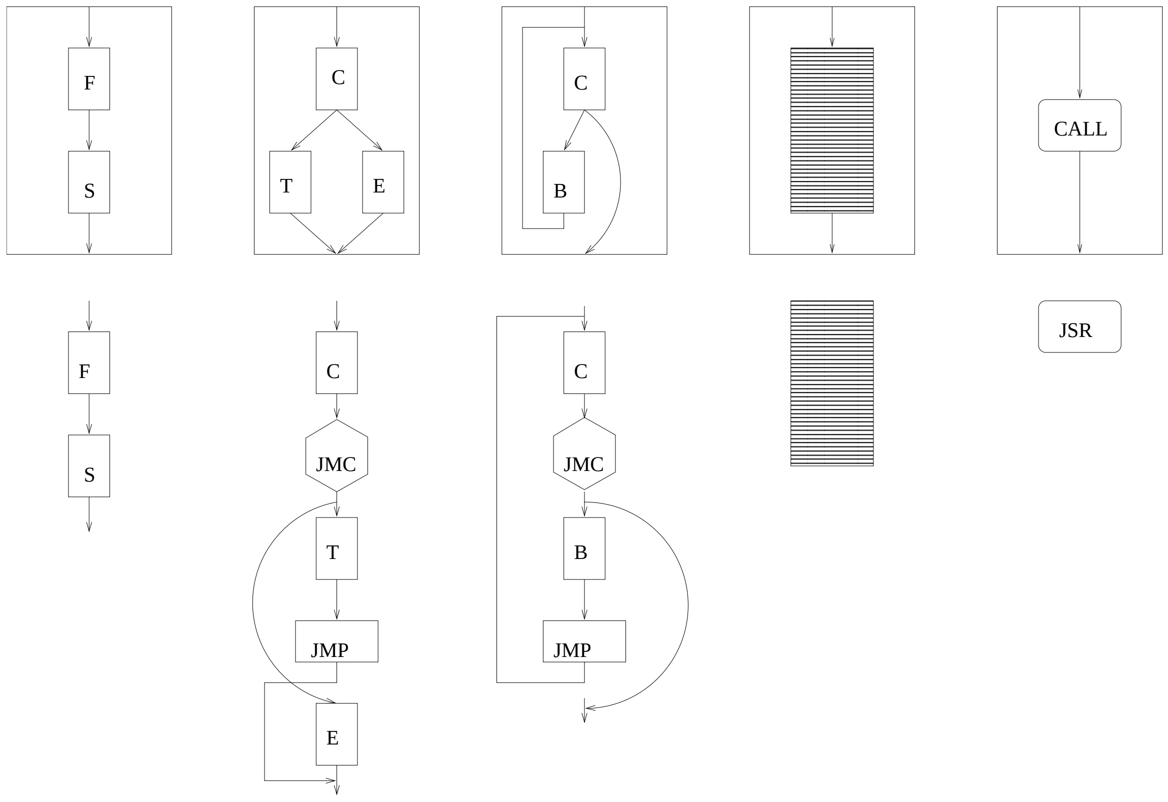

Figure 5: Linearization of Basic Blocks

```
% --- linearization of basic blocks ---                                    55
lin(g:bb_graph) : RECURSIVE linear_code =
    CASES g OF
     simple_block(p)    : (: lin_code(p) :),
     seq_block(b1,b2)   : lin(b1) ++ lin(b2),
     if_block(c,t,e)    : LET bb = lin(c), l1 = lin(t), l2 = lin(e) IN
                           (bb ++
                           (: jmc(length(l1) + 2) :) ++
                            l1 ++
                            (: jmp(length(l2) + 1) :) ++
                            l2),
     while_block(c,b)   : LET bb = lin(c), l = lin(b) IN
                          (bb ++
                           (: jmc(length(l) + 2) :) ++
                           l ++
                           (: jmp(-(length(l) + length(bb) + 1)) :)),
     call_block(k)      : (: jsr(k) :)
   ENDCASES
    MEASURE g BY <<

%%% compilation of basic block programs
 cp(p:bb_program): machine_program =
   (# sdecls := LAMBDA (i:PId): lin(pd(p)(i)), main := lin(main_block(p)) #)
```

Informally this means: a simple block consisting of code sequence `p` is translated into a `lin_code(p)` instruction, sequential blocks are linearized by recursively linearizing the subblocks and composing the resulting code sequences. For the if-block and while-block there are several possibilities to linearly order the subblocks. For the if-block we have selected the sequential order `if_cnd, thn, els` by introducing a conditional jump after the code for `if_cnd` block and an unconditional jump to the end of the sequence after the code for `thn`. The linearization of the while-block consists of linearizing both the `while_cnd` and `body` subblock together with a conditional jump and an unconditional jump back to the beginning of the sequence. Finally, a call block is tranlated into a corresponding jump subroutine instruction.

The correctness of this compilation step is stated by the following theorem and is illustrated in Fig. 6, an instance of the diagram in Fig. 2. Since both the semantics of basic blocks and machine programs are defined by state transformers on `MState` source and target language states correspond. `ID` denotes the identity on `MState`. Here, *semantic equivalence* is established: the semantics of a block program `g` is equal to the semantics of the associated machine program.

```
linearization_correct : THEOREM FORALL (p: bb_program): Ip(cp(p)) = BS(p)
```

We split the proof of `linearization_correct` into two parts: we show

-
  ```
  linearization_correct1 : THEOREM Ip(cp(p)) <= BS(p)
  ```

-
  ```
  linearization_correct2 : THEOREM BS(p) <= Ip(cp(p))
  ```

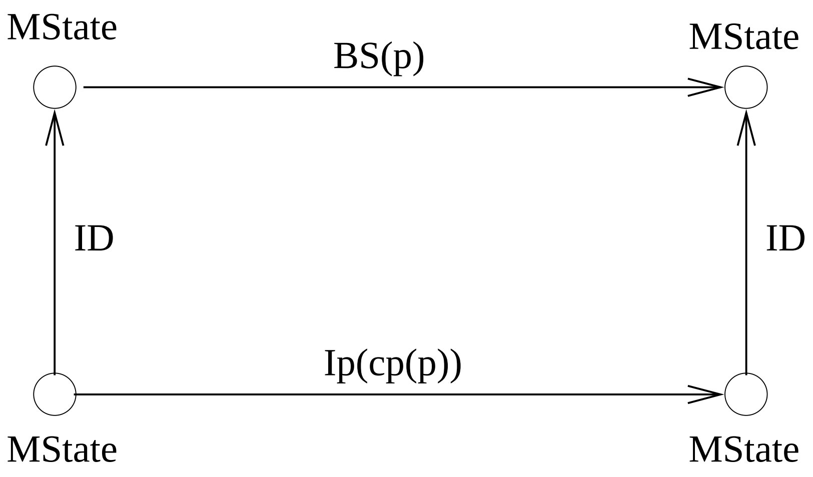

Figure 6: Correctness of Linearization Step

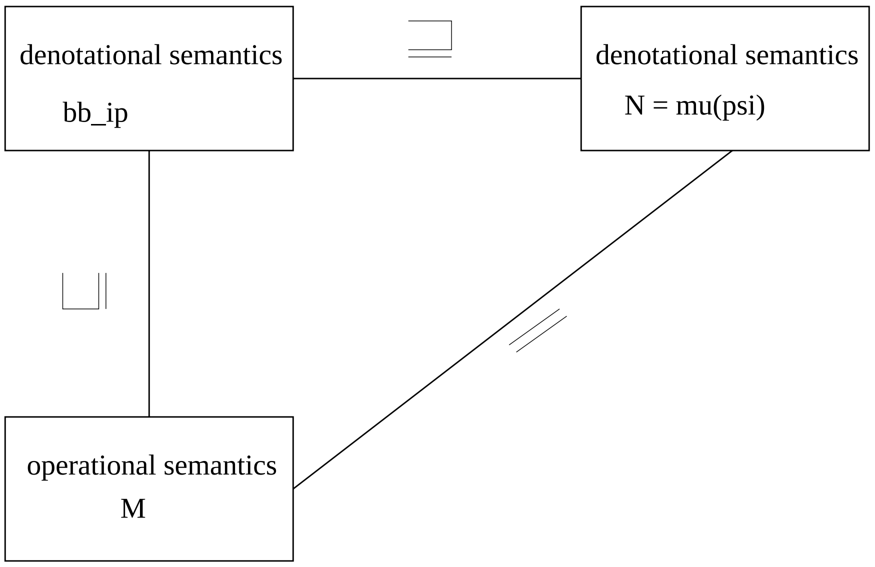

Figure 7: Denotational Characterization of Machine Semantics

7.2.1 Proof of `linearization_correct1`

Consider first `linearization_correct1`. The principle idea of the proof is to derive a denotational characterization of the (operational) machine semantics as the least fixed-point of a functional `psi` and then to apply fixed-point induction. Fig. 7 illustrates the proof idea.

A Denotational Machine Semantics

Note, that the semantics of machine programs (the interpreter) `Ip` is defined using relation `M` $\boxed{48}$. A denotational characterization `N` of `M` is accomplished by showing that `M` is the smallest function satisfying the characteristic laws in $\boxed{49}$ - $\boxed{54}$.

```
ind_type : TYPE = [machine_program -> [Stack -> srel]]
% h : VAR ind_type

psi(h)(mp)(cf): srel =
 IF empty?(cf) THEN skip
 ELSE
  LET t = top(cf), code = proj_1(t), n = proj_2(t) IN
  IF (n < 1 OR n > length(code)) THEN h(mp)(pop(cf))    % return
   ELSE
      CASES nth(code, n - 1) OF
        lin_code(p):  interprete(p) ++ h(mp)(push((code, n + 1), pop(cf))),
        jmp(i):       h(mp)(push((code, n + i), pop(cf))),
        jmc(i):       fork(h(mp)(push((code, n + 1), pop(cf))),
                           h(mp)(push((code, n + i), pop(cf)))),
        jsr(k):       h(mp)(push((sdecls(mp)(k), 1), push((code, n + 1), pop(cf))))
      ENDCASES
     ENDIF
  ENDIF

psi_monotonic: LEMMA monotonic?[ind_type,ind_type](<=,<=)(psi)
```

To define the least fixed-point of `psi`, a cpo over `ind_type` has to be defined. This is done by repeatedly using the cpo constructor `=>`:

```
conf_th: THEORY = FP@exponent[Stack,srel]
conf_cpo : CPO[[Stack -> srel]] = (discrete[Stack] => srel)

ind_th: THEORY = FP@exponent[machine_program, [Stack -> srel]]
ind_cpo: CPO[[machine_program -> [Stack -> srel]]] =
 (discrete[machine_program] => conf_cpo)

; <= : preCPO[ind_type] = (<=(ind_cpo))
bottom:Bottom[ind_type](<=) = (bottom(ind_cpo))

%%% denotational semantics N
N: ind_type = mu(ind_cpo)(psi)
```

The existence of the least fixed-point of `psi` is proved using `psi_monotonic` and `KnasterTarski`.

We have to establish the equivalence of the operational and denotational machine semantics $\boxed{56}$.

$\boxed{56}$
```
sem_equivalence : THEOREM N = M
```

Again, the proof is split into two parts:

```
N_leq_M: LEMMA N <= M
M_leq_N: LEMMA M <= N
```

Since `psi` is the smallest function satisfying the characteristic laws of `M`, `M` is a fixed-point of `psi` $\boxed{57}$.

$\boxed{57}$
```
M_is_fp: LEMMA psi(M) = M
```

The proof of $\boxed{57}$ is by a case analysis and using the laws $\boxed{49}$ - $\boxed{54}$.

For the other direction we show that configuration transitions are correctly reflected by the denotational semantics. First, this is proved for a one-step transition $\boxed{58}$. The proof is by a case analysis on the type of instruction and unfolding definitions.

$\boxed{58}$
```
eff_den_correct : LEMMA
  eff(mp)((ms1,s1), (ms2,s2))
   IMPLIES N(mp)(s1)(ms1) = N(mp)(s2)(ms2)
```

Then, analogously, a n-step transition produces the same effect for `N` $\boxed{59}$. This is proved by rule induction using `RTC_induction` and `eff_den_correct`.

$\boxed{59}$
```
RTC_eff_den_correct : LEMMA
 RTC(eff(mp))((ms1,s1), (ms2,s2))
  IMPLIES N(mp)(s1)(ms1) = N(mp)(s2)(ms2)
```

We are now able to proceed with the proof of `linearization_correct1`. In order to establish this proof a stronger property is used. This is necessary because when proving the induction step for sequential blocks one cannot generally conclude that the program counter points at the beginning of the second part after the first part is evaluated. The idea is to introduce a code context `u,v` around the program of interest $\boxed{60}$.

$\boxed{60}$
```
prop(p:bb_program)(h:ind_type) : bool =
 FORALL g: FORALL a,u,v:
    h(cp(p))(push((u ++ lin(g) ++ v, 1 + length(u)), a))
     <= bb_ip(g)(D(pd(p))) ++
         h(cp(p))(push((u ++ lin(g) ++ v, 1 + length(u) + length(lin(g))), a))

%%% auxiliary property
prop1(p:bb_program)(h:ind_type)(g) : bool =
 FORALL a,u,v:
    h(cp(p))(push((u ++ lin(g) ++ v, 1 + length(u)), a))
     <= bb_ip(g)(D(pd(p))) ++
         h(cp(p))(push((u ++ lin(g) ++ v, 1 + length(u) + length(lin(g))), a))
```

Informally, this property states that if program u ++ m ++ v (the body of the current procedure) is executed with the first instruction of m where m is the result of the linearization step then the behavior corresponds to the state transformer given by the semantics of graph g composed with the state transformer given by the machine program semantics starting the machine program at the first instruction of v. The main challenge is therefore to show that for all block graphs g theorem $\boxed{61}$ holds.

```
correct: LEMMA FORALL p: prop(p)(N)                                    61
```

From this theorem, the main conjecture linearization_correct1 can easily be derived by instantiating the empty code sequence for the context u and v, the main program for g, the empty stack for a, and applying law_identity $\boxed{49}$.

Since N is defined as the least fixed-point of functional psi the proof of $\boxed{61}$ is established using fixed-point induction. Here, we use the induction principle fp_induction_mono_le, see $\boxed{1}$.

One has to prove

- the admissibility of our conjecture:

  ```
  prop_admissible: LEMMA                                                62
   admissible?[ind_type](PROJ_1(ind_cpo))(LAMBDA h: FORALL p: prop(p)(h))
  ```

  As in the admissibility proof sketched in Section 6, the proof idea here is to characterize the least upper bound of chain C of functions of type ind_type as the function mapping a linear program, an integer, and a state to the union of all images of the functions in the chain C.

- the induction base, (trivial), and

- the induction step

The induction step is proved by structural induction on the construction of basic block graphs g. Trying to keep the proof effort manageable separate theorems for each block constructor have been established in $\boxed{63}$. Using these compilation theorems the proof of the induction step is accomplished easily.

```
%%% compilation theorems %%%                                             63

correct_si_lin1 : LEMMA
 h <= psi(h) IMPLIES prop1(p)(psi(h))(simple_block(c))

correct_seq_lin1 : LEMMA
  prop1(p)(psi(h))(b1) & prop1(p)(psi(h))(b2)
   IMPLIES prop1(p)(psi(h))(seq_block(b1,b2))

correct_itef_lin1 : LEMMA
 prop1(p)(psi(h))(b1) & prop1(p)(psi(h))(b2) & prop1(p)(psi(h))(b3) & h <= psi(h)
  IMPLIES prop1(p)(psi(h))(if_block(b1,b2,b3))

correct_while_lin1 : LEMMA
  prop1(p)(psi(h))(b1) & prop1(p)(psi(h))(b2) &
   prop1(p)(h)(while_block(b1,b2)) & h <= psi(h)
    IMPLIES prop1(p)(psi(h))(while_block(b1,b2))

correct_call_lin1: LEMMA
  prop1(p)(h)(pd(p)(i)) & h <= psi(h)
    IMPLIES prop1(p)(psi(h))(call_block(i))
```

The proofs of the compilation theorems are by unfolding of definitions and rewriting using some monotonicity and associativity properties for sequential composition. The proofs of the first two lemmas are relatively easy to accomplish, the proofs consist of approximately 10 PVS proof steps. The proofs of the if-block and while-block theorems require more effort (approx. 50 interactions) since a lot of monotonicity properties have to be exploited.

7.2.2 Proof of `linearization_correct2`

For this proof the operational semantics `M` is used directly. As above, we prove a stronger property by adding a code context `u, v` around the code of interest $\boxed{64}$.

```
prop(p:bb_program)(rho:environment) : bool =                             64
 FORALL (g:bb_graph): FORALL  (a:Stack), (u,v:linear_code):
    bb_ip(g)(rho) ++
     M(cp(p))(push((u ++ lin(g) ++ v, 1 + length(u) + length(lin(g))), a))
      <= M(cp(p))(push((u ++ lin(g) ++ v, 1 + length(u)), a)))

prop_aux(p)(rho)(g): bool =
 FORALL (a:Stack), (u,v:linear_code):
    bb_ip(g)(rho) ++
     M(cp(p))(push((u ++ lin(g) ++ v, 1 + length(u) + length(lin(g))), a))
      <= M(cp(p))(push((u ++ lin(g) ++ v, 1 + length(u)), a)))
```

One has to prove

```
main_obligation2 : THEOREM FORALL p: prop(p)(D(pd(p)))                   65
```

The proof is by fixed-point induction on the procedure environment. Note that the declaration semantics `D` is defined as a least fixed-point. Again, the induction base is trivial.

Admissibility of the property is proved as above. The induction step is by structural induction on `g`, and compilation theorems have been introduced in $\boxed{66}$:

$\boxed{66}$
```
%%% compilation theorems %%%
correct_si_lin2 : LEMMA prop_aux(p)(rho)(simple_block(q))

correct_seq_lin2 : LEMMA
 prop_aux(p)(rho)(b1) & prop_aux(p)(rho)(b2)
  IMPLIES prop_aux(p)(rho)(seq_block(b1,b2))

correct_itef_lin2 : LEMMA
 prop_aux(p)(rho)(b1)  & prop_aux(p)(rho)(b2) & prop_aux(p)(rho)(b3)
  IMPLIES prop_aux(p)(rho)(if_block(b1,b2,b3))

correct_while_lin2 : LEMMA
 prop_aux(p)(rho)(b1) & prop_aux(p)(rho)(b2)
  IMPLIES prop_aux(p)(rho)(while_block(b1,b2))

correct_call_lin2: LEMMA
 prop(p)(rho) IMPLIES prop_aux(p)(decl_sem(pd(p))(rho))(call_block(i))
```

The proofs of the first three compilation lemmas are relatively easy to accomplish using the characteristic laws for `M` and some monotonicity properties for sequential composition and `fork`. The while theorem requires the transfer lemma $\boxed{9}$, and park's lemma.

Using `main_obligation2` $\boxed{65}$, instantiating the empty code sequence for `u` and `v` in $\boxed{64}$, the main program for `g`, the empty stack for `a`, and applying `law_identity` $\boxed{49}$ finishes the proof of `linearization_correct2` and we are done.

8 Implementation of Procedures

Upto now, an (abstract) machine program consists of procedures and a main program where the procedure bodies and main program consist of linear code (with jumps and subroutine calls).

$$p_1 \leftarrow c_1 \ldots, p_n \leftarrow c_n \ ; \ main$$

The semantics is defined using a stack of tuples consisting of a linear code (denoting the body of the called subroutine) and local program pointer. In this compilation step, procedures are linearized and the semantics makes use of jump tables instead of abstract stacks. The step is carried out in two phases (refinement steps):

- first, an explicit `return` instruction is introduced, and the (abstract) machine is refined.

$$p_1 \leftarrow c_1 \circ ret, \ldots, p_n \leftarrow c_n \circ ret \ ; \ main \circ ret$$

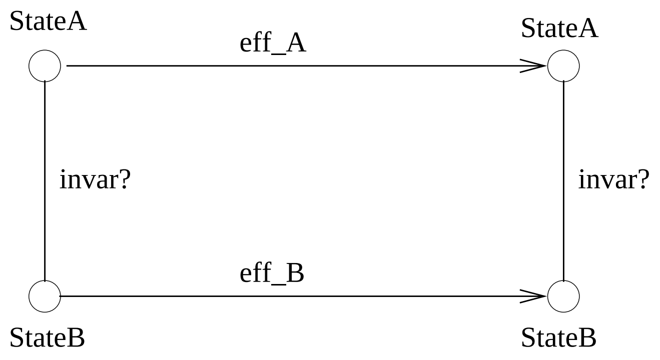

Figure 8: One-Step Simulation

- then, the procedures and main program are linearly ordered, and the (abstract) machine is again refined.

$$c_1 \circ ret \circ c_2 \circ ret \circ \ldots \circ c_n \circ ret \circ main \circ ret$$

8.1 Introduction of a Return Instruction

A slightly modified abstract machine is specified where an additional return instruction is introduced. We do not repeat all definitions here since they are exactly the same as described in section 7.1. We therefore concentrate on the differences.

```
%%% additional return instruction                                    67
 MInstr : DATATYPE
   BEGIN
     lin_code(get_ins : Code) : lc?
     jmp(jp_adr:int)  : jmp?
     jmc(jc_adr:int)  : jmc?
     jsr(jsr_adr:PId) : jsr?
     ret              : ret?
   END MInstr
```

However, the configuration one-step semantics `eff` is more restricted for the new machine. In case, the current program pointer points somewhere outside the actual procedure body, there is no successor state. Hence, the modified abstract machine is a refinement of the old one.

A mapping is defined from the abstract machine to the modified abstract machine. Each instruction is compiled one-to-one to its corresponding instruction on the modified machine. At the end of a procedure body and the main program, a return instruction is introduced.

To establish correctness of this compilation step, a classical simulation proof for abstract machines is required. First, the correspondence of a single step is proved (Fig. 8).

$$invar? \; ; \; \mathit{eff}_B(compile(mp)) \;\subseteq\; \mathit{eff}_A(mp) \; ; \; invar?$$

Then, by rule induction, one can prove that this holds also for the reflexive, transitive closure. Here, the invariant between abstract and concrete states is given by a correspondence of the stacks, and equality of the abstract states (of type `MState`):

```
%%% invariant between abstract and concrete states
eq_stack(mp)(s1,s2) : RECURSIVE bool =
  depth(s1) = depth(s2) AND
     (not(empty?(s1)) IMPLIES
       proj_2(top(s1)) = proj_2(top(s2)) AND
        proj_1(top(s2)) = compileProc(proj_1(top(s1))) AND
         eq_stack(mp)(pop(s1), pop(s2)))
     MEASURE depth(s1)

invar?(mp)(c1:l0.Conf, c2:l1.Conf): bool =
 proj_1(c1) = proj_1(c2) AND eq_stack(mp)(proj_2(c1), proj_2(c2))
```

The correspondence of the single effects `eff` is stated by

```
effect_simul: LEMMA
 invar?(mp)(a1,b1) AND
   eff(compile(mp))(b1,b2)
     IMPLIES EXISTS a2: eff(mp)(a1,a2) AND invar?(mp)(a2,b2)
```

The proof is not difficult but lengthy since there are many cases to consider. Using rule induction the correspondence for the reflexive, transitive closure can be proved:

```
tc_simul1: LEMMA
  RTC(eff(compile(mp)))(b1,b2) AND invar?(mp)(a1,b1)
    IMPLIES EXISTS a2: RTC(eff(mp))(a1,a2) & invar?(mp)(a2,b2)

tc_simul: LEMMA
 RTC(eff(compile(mp)))((ms1, s2), ((ms2, empty))) AND
   invar?(mp)((ms1, s1), (ms1, s2))
     IMPLIES RTC(eff(mp))((ms1, s1), ((ms2, empty)))
```

Finally, one can conclude that the modified machine is a refinement of the old one:

```
machine_simulation: THEOREM Ip(compile(mp))(ms1)(ms2) IMPLIES Ip(mp)(ms1)(ms2)
```

8.2 Linearization of Procedures and Main Program

The final step is to linearize the procedures and the main program. This is realized, by

- introducing an additional jump table for start and final procedure addresses, and the start and final address of the main program

- changing the configurations:
  - "old" configurations use stacks of tuples of actual procedure and *local* program pointer,
  - the modified configurations use stacks of triples of start, final and return addresses of the actual procedure and a *global* program pointer

68

```
%%% machine programs
mprg: TYPE =
 [# program:linear_code, stab: jumptable, ftab: jumptable, start: nat, final:nat #]

%%% configurations
Conf : TYPE = [MState, Stack[[# startadr: nat, finaladr:nat, retadr:nat #]], int]
```

The single-step configuration semantics is as follows:

69

```
eff(mp)(c1,c2) : bool = LET (ms1, s1, pc1) = c1, (ms2, s2, pc2) = c2 IN
 IF empty?(s1) THEN false
  ELSE
   IF (pc1 >= finaladr(top(s1)) OR
        pc1 < startadr(top(s1)) OR pc1 >= length(program(mp))) THEN false
   ELSE CASES nth(program(mp), pc1) OF
      lin_code(p)  :  interprete(p)(ms1)(ms2) & s2 = s1 & pc2 = pc1 + 1,
      jmp(i)       :  ms2 = ms1 & s2 = s1 & pc2 = pc1 + i,
      jmc(i)       :  IF outputdefd?(ms1) THEN
                       IF output(ms1) THEN
                         ms2 = read(ms1) & s2 = s1 & pc2 = pc1 + 1
                       ELSE
                         ms2 = read(ms1) & s2 = s1 & pc2 = pc1 + i
                       ENDIF
                       ELSE false
                      ENDIF,
      jsr(j)       :  LET newtop = (# startadr := stab(mp)(j),
                                      finaladr := ftab(mp)(j),
                                      retadr := pc1 + 1 #)
                      IN
                       ms2 = ms1 & s2 = push(newtop, s1) & pc2 = stab(mp)(j),
      ret          :  IF empty?(s1) THEN false
                      ELSE ms2 = ms1 & s2 = pop(s1) & pc2 = retadr(top(s1))
                      ENDIF
     ENDCASES
   ENDIF
 ENDIF
```

`M` is defined as the reflexive, transitive closure of `eff` such that the final stack is empty:

70

```
M(mp)(s1)(pc1)(ms1)(ms2) : bool =
 EXISTS pc2: RTC(eff(mp))((ms1, s1, pc1), (ms2, empty, pc2))
```

The semantics of machine programs is then defined by the interpreter `Ip`:

```
starttop(mp) : stackelem =                                                   71
                  (# startadr := start(mp),
                     finaladr := final(mp),
                     retadr   := 0 #)

Ip(mp) : strans = M(mp)(push(starttop(mp), empty))(start(mp))
```

The translation of machine programs is as follows:

- instructions are translated one-to-one.
- the body codes of the procedures are concatenated, and the main program is appended at the end
- start and final addresses of the procedures are calculated
- start and final addresses of the main program are calculated

To establish correctness, one has to deal with the calculation of sublists. For this purpose, a function for list extraction together with a set of properties has been established:

```
%%% extraction function                                                      72
%%% extract(x)(i,j) = x.i ... x.(j-1) for j > i and i < length(x)

extract(l:list[T])(i,j:nat): RECURSIVE list[T] =
 IF (j <= i OR i >= length(l)) THEN null[T]
 ELSE cons(nth(l, i), extract(l)(i + 1, j))
 ENDIF
  MEASURE IF j >= i THEN j - i ELSE 0 ENDIF

extract_p5: LEMMA (m <= length(x)) IMPLIES extract(append(x,y))(n,m) = extract(x)(n,m)
...
```

The invariant between abstract and concrete configurations requires that

- abstract states `MState` are equal
- global and local program pointer refer to the same instruction, and
- abstract and concrete stacks correspond

Correspondence on the stacks is specified by the predicate:

73
```
eq_stack(mp)(s1,s2) : RECURSIVE bool =
 depth(s1) = depth(s2) AND
 (not(empty?(s1)) IMPLIES
   not(empty?(s2)) AND LET (mc, pc1) = top(s1) IN
     startadr(top(s2)) < length(program(compile(mp))) AND
     finaladr(top(s2)) <= length(program(compile(mp))) AND
     extract(program(compile(mp)))((startadr(top(s2)), finaladr(top(s2)))) =
      compileC(mc) AND
          ( not(empty?(pop(s1))) IMPLIES
            (not(empty?(pop(s2))) AND
             proj_2(top(pop(s1))) + startadr(top(pop(s2))) = retadr(top(s2)) + 1)) AND
             eq_stack(mp)(pop(s1), pop(s2)))
  MEASURE depth(s1)
```

The invariant is given by

74
```
invar?(mp)(c1:lc1.Conf, c2:lc2.Conf): bool =
 proj_1(c1) = proj_1(c2) AND                  %%% abstract states (MState) are equal
  (not(empty?(proj_2(c1))) IMPLIES            %%% pc's correspond
    not(empty?(proj_2(c2)))
      AND startadr(top(proj_2(c2))) + proj_2(top(proj_2(c1))) = proj_3(c2) + 1) AND
       eq_stack(mp)(proj_2(c1), proj_2(c2))  %%% stacks correspond
```

The proof obligations are the same as presented in the last subsection. First, one-step correspondence has to be established, then, by rule induction, the correctness of the closures is proved. The final result is that the modified machine is a refinement of the "old" one:

75
```
%%% main result
machine_simulation: LEMMA Ip(compile(mp))(ms1)(ms2) IMPLIES Ip(mp)(ms1)(ms2)
```

9 Specific Compilation Processes

In order to illustrate the applicability of the generic compilation theories two specific compilation processes are presented. In particular, we describe the compilation of a simple imperative language consisting of expressions and statements into code of a

- stack machine, and a
- one-address accumulator machine.

We start with defining syntax and semantics of our simple imperative language. For defining syntax and semantics of expressions the parameters in 76 are used abstracting from the concrete type of expression values and from the available set of unary and binary operators.

```
% --- parameters used for specifying the source language ---                76
VarId   : TYPE,
PId     : TYPE+,
Value   : TYPE,
Unop    : TYPE,
Binop   : TYPE,
MUnop   : [Unop   -> [Value -> Value]],
MBinop  : [Binop  -> [Value,Value -> Value]]
```

Value denotes the type of source values, VarId the type of identifiers, (Unop, Binop) the available set of unary and binary operators and their semantics (MUnop) resp. (MBinop). Abstract datatype Expr and an evaluation function eval ($\boxed{77}$) then define syntax and semantics of expressions where the state (SState) is defined as a mapping from identifiers to values.

```
% --- semantics of expressions ---                                          77
eval(e:Expr)(s:SState) : RECURSIVE Value =
  CASES e OF
   const(val)             : val,
   varid(name)            : s(name),
   unopr(op,arg)          : MUnop(op)(eval(arg)(s)),
   binopr(op,left,right)  : MBinop(op)(eval(left)(s), eval(right)(s))
  ENDCASES
   MEASURE e BY <<
```

Since boolean expressions are treated in a similar way as expressions, we do not define them explicitly but instead suppose that an (uninterpreted) type BExp together with an evaluation function eval_bexp : [BExp -> [SState -> bool]] is given.

Syntax and semantics of statements are defined by importing the generic theories for simple statements and control structures:

```
% --- import syntax and semantics of simple statements
IMPORTING simple_statements[VarId, Expr, Value, eval]

% --- import syntax and semantics of control structures
IMPORTING ctrlstruc[BExp, SState, PId, eval_bexp, SimpleStatement, ss_meaning]
```

In the following subsection we deal with the compilation of this language into stack machine code, then in Section 9.2 its compilation into code of a one-address machine is described. For both machines, compilation of expressions is outlined explicitly while compilation of statements is carried out by instantiating the generic theories.

9.1 A Stack Machine Compilation

We consider a stack machine which is parameterized with respect to the type of memory addresses, the type of machine values, and the set of available unary and binary ALU operations and their semantics. It includes instructions for

- loading a literal onto the stack (LIT),

- loading the contents of a specific memory cell onto the stack (LOAD),
- applying unary and binary operators (UNOP, BINOP),
- and storing the stack's top element into memory (STORE).

The memory is a mapping from addresses to values, and the machine state consists of the stack and the memory combined in a record type `MachineState`.

```
MachineState : TYPE+ = [# stack: Stack, mem: Mem #]
```

The effects of each instruction are specified by function `onestep` in $\boxed{78}$.

```
litf(v:Value)(s) : (deterministic?) =                                          78
 singleton(s WITH [(stack) := push(v,stack(s))])

loadf (a:Addr)(s) : (deterministic?) =
 singleton(s WITH [(stack) := push(mem(s)(a),stack(s))])
...

onestep(i:Instr) : PartialFunction[MachineState,MachineState] =
 CASES i OF
  LIT(v)    : litf(v),
  LOAD(a)   : loadf(a),
  UNOP(op)  : uopf(op),
  BINOP(op) : bopf(op),
  STORE(a)  : storef(a)
 ENDCASES
```

For defining the semantics of a code sequence the generic interpreter is imported:

```
IMPORTING simple_interpreter[Instr,MachineState,onestep]                       79
```

Compilation

Consider now the compilation of expressions into stack machine code. We suppose given a predicate `representable?` denoting the set of source values which are representable on the target architecture. The compilation function $\boxed{80}$ may only compile constants which have representable values. We further suppose that a bijection `valmap` from target values to representable source values is given. In addition, an injective function `idmap` mapping identifiers to memory addresses is required.

```
compile(e:Expr) : RECURSIVE (deterministic?[Code]) =                          80
    CASES e OF
      const(val)          :
                        IF representable?(val)
                          THEN singleton((: LIT(inverse(valmap)(val)):Instr :))
                          ELSE emptyset
                        ENDIF,
      varid(name)         : singleton((: LOAD(idmap(name)):Instr :)),
      unopr(op,arg)       : compile(arg) ++ singleton((: UNOP(op):Instr :)),
      binopr(op,left,right) : compile(left) ++
                              compile(right) ++ singleton((: BINOP(op):Instr :))
    ENDCASES
      MEASURE e BY <<
```

Correctness of compilation is stated using predicate `correct` in $\boxed{81}$. An abstraction function `statemap` mapping machine states to program states is defined using `valmap` and `idmap`.

```
statemap(ms:MachineState) : SState =                                           81
 LAMBDA (v:VarId): valmap(mem(ms)(idmap(v)))

correct(e:Expr,c:Code) : bool =
  FORALL (start,final:MachineState):
   interprete(c)(start)(final)
     IMPLIES
      nonempty?(stack(final)) AND
       eval(e)(statemap(start)) = valmap(top(stack(final))) AND
        statemap(final) = statemap(start)
```

To establish correctness of expression compilation, one has to prove:

```
correctness : THEOREM compile(e)(c) IMPLIES correct(e,c)                       82
```

The proof is by induction on the structure of `e`. The base cases for constants and identifiers as well as the induction step for unary operators can be proved easily. To prove the induction step for binary operators, one first has to establish an invariant `interprete_invariant` which states that when interpreting the compiled code in the final state the stack contains an additional element. This ensures that executing code for the second subexpression does not effect the value of the first one, i.e. the value of the first subexpression is preserved. The proof of the invariant is also by structural induction on `e`.

```
interprete_invariant : LEMMA                                                   83
 compile(e)(c) AND interprete(c)(start)(final)
  IMPLIES
   EXISTS (v:TarValue): stack(final) = push(v, stack(start))
```

Consider now compilation of statements. In order to utilize the generic compilation theory for simple statement compilation described in Section 5, specific values must be provided for the abstract parameters. More specifically,

- The output function accesses the top element of the stack. Hence, the output is only defined in states in which the stack contains at least one element. Since the access location of values is constant for this machine (top of stack), parameter `T` in the generic theory is not required and instantiated with a default type (`unit`) consisting of exactly one element (`one`).

```
% --- access of values ---

outputdefd?(u:unit)(ms:MachineState) : bool = nonempty?(stack(ms))
output(u:unit)(ms:(outputdefd?(u))) : TarValue = top(stack(ms))
```

- Code for storing values into memory at a specific address is given by the single `STORE` instruction:

```
% --- storing values ---
STORE_code(u:unit, a:Addr) : Code = (: STORE(a):Instr :)
```

- To match the signature of the parameter `compileExpr` we simply extend the compilation function `compile` as follows:

```
% --- compilation of expressions ---
compileExpr(e) : [(deterministic?[Code]), unit] = (compile(e), one)
```

Using these definitions, and let `target_memory` denote the state record selector `mem`, the generic theory can be imported.

```
% --- import compilation of simple statements ---
compile_assign  [VarId, Expr, Value, eval, Instr, MachineState,
                 onestep, Addr, TarValue, unit, outputdefd?,
                 RegFile, STORE_code, target_memory,
                 representable?, valmap,
                 idmap, compileExpr, statemap]
```

Importing this theory, four assumptions are generated, see $\boxed{16}$, $\boxed{18}$, and $\boxed{19}$. Assumption `expression_compilation_correct` is discharged using theorem `correctness` above. Assumption `interprete_store` is proved easily by unfolding definitions. Assumption `symtab_and_memory` is trivial, and finally the proof of assumption `statemap_and_memory` requires injectivity of `idmap`.

Consider now compilation of control structures. In order to use the generic theory for compiling control structures into basic blocks, a `read` function has to be defined. The read function for stack machine is simply a pop operation on the current stack. In addition, an output function with range type `bool` is required.

```
% --- access of truth values ---                                          84

read(msdef:(outputdefd?)) : MachineState = msdef WITH [(stack) := pop(stack(msdef))]
output_bool(msdfd:(outputdefd?)) : bool
```

We will not consider compilation of boolean expressions explicitly since it closely follows the compilation of expressions. We suppose given a compilation function `compileBExpr` for boolean expressions satisfying a correctness assumption `bexp_comp_correct`:

```
compileBExpr(b:BExp) : (deterministic?[Code])

bexp_comp_correct : AXIOM
   (FORALL (b:BExp, c:Code):
     compileBExpr(b)(c) IMPLIES
      FORALL (start,final: MachineState):
       interprete(c)(start)(final) IMPLIES
         nonempty?(stack(final)) AND
           eval_bexp(b)(statemap(start)) = output_bool(final) AND
             statemap(final) = statemap(start))
```

Importing the generic theory for compiling control structures into basic blocks, three assumptions have to be proved. It must be proved that (boolean) expression compilation and simple statement compilation are correct. Using the axiom above and the generic compilation theorem `simple_statement_comp_correct` $\boxed{22}$, respectively, these obligations can be discharged easily. The third obligation states that `read` must have no effects on corresponding source states. It is proved automatically using `GRIND`. Finally, the generic theory for linearization is imported. Since this theory does not contain assumptions no proof obligations are generated. Finally, the following theorem which states correctness of basic block compilation and linearization of control structures can be proved easily using the generic theorems `correctness` and `linearization_correct`:

```
% --- compilation of source programs is correct                              85
stmts_compile_correct: THEOREM FORALL (p:source_program):
 FORALL (start,final:MachineState):
   compile_defined?(p) AND
     Ip(cp(compile(p)))(start)(final)
       IMPLIES P(p)(statemap(start))(statemap(final))
```

9.2 Compilation into a One-Address Machine

Our simple one-address machine is parameterized in the same way as the stack machine described in the last subsection. `Addr` denotes the type of memory addresses, `MValue` the type of values, `Unop, Binop, munop_sem, mbinop_sem` the available set of unary and binary operators and their semantics. Here, the machine state consists of an accumulator, the memory (a mapping from addresses to values), and a flag of type `bool`. The machine does not contain general registers. There are instructions for

- loading a literal into the accumulator (`LIT`),
- loading the contents of a specific memory cell into the accumulator (`LOADA`),
- applying unary and binary operators (`UNOP, BINOPA`),
- and storing the content of the accumulator into memory (`STOREA`).

```
MState : TYPE+ = [# ac:MValue, mem: [Addr -> MValue], flag:bool #]
```

The effects of each instruction are specified by function `one_step` in 86; all arithmetic operations are carried out using the accumulator.

```
% --- effects of instructions ---                                          86
one_step(i:Instr) : PartialFunction[MachineState,MachineState] =
 CASES i OF
  SETFLAG(flg)  : singleton(ms WITH [(flag) := flg]),
  LIT(v)        : singleton(ms WITH [(ac) := v]),
  LOADA(a)      : singleton(ms WITH [(ac) := mem(ms)(a)]),
  UNOP(uop)     : singleton(ms WITH [(ac) := munop_sem(uop)(ac(ms))]),
  BINOPA(bop,a) : singleton(ms WITH
                          [(ac) := mbinop_sem(bop)(ac(ms), mem(ms)(a))]),
  STOREA(a)     : singleton(ms WITH [(mem) := mem(ms) WITH [(a) := ac(ms)]])
 ENDCASES
```

For defining the semantics of a code sequence the generic interpreter is imported:

```
IMPORTING simple_interpreter[Instr,MachineState,onestep]                   87
```

Compilation

As for the stack machine compilation, we suppose given a predicate `representable?` denoting the set of representable source values, a bijection `valmap` from target values to representable source values, and an injective memory mapping `idmap` from identifiers to target addresses. Since this machine does not have a stack mechanism, temporary locations for storing intermediate values have to be allocated. More specifically, compiling a binary expression `bop(e1,e2)` consists of first generating code for `e2`, saving this value into a temporary location, then generating code for `e1` and code for the operator `bop` which then accesses the values from the temporary location and the accumulator. The compilation of expressions thus starts with a set of available temporary locations from which required temporaries are taken. If there are not enough temporaries the compilation function is undefined, i.e. returns the empty code set. Type `tempset` specifies the type of such a set. It is required that locations onto which identifiers are mapped by `idmap` are not used as temporaries. For allocating locations we suppose given a function `ralloc` which selects a free location from a (nonempty) set of temporaries.

```
tempset:TYPE = {M:set[Addr] | FORALL (id:Ident): not(member(idmap(id), M))}
```

The complete compiling function is given by `compile` in 88

88
```
% --- compilation of expressions ---
% RT : TYPE = [Code, tempset]

compile(e:Expr)(free:tempset) : RECURSIVE (deterministic?[RT]) =
 CASES e OF
  const(val)     : IF not(representable?(val)) THEN emptyset[RT]
                   ELSE singleton[RT](((: LIT(inverse(valmap)(val)) :):Code, free))
                   ENDIF,
  varid(name)    : singleton[RT](((: LOADA(idmap(name)) :):Code, free)),
  unopr(unop,e1) : LET m = compile(e1)(free) IN
                    IF empty?[RT](m) THEN emptyset[RT]
                    ELSE LET (code, rest) = select(m) IN
                      singleton[RT]((code ++ (: UNOP(unop) :), free)),
                    ENDIF,
  binopr(bop,e1,e2) :
   LET m2 = compile(e2)(free) IN
    IF empty?[RT](m2) THEN emptyset[RT]
    ELSE LET (code_e2, free_e2) = select(m2) IN
          IF empty?[Addr](free_e2) THEN emptyset[RT]
          ELSE LET temp = ralloc(free_e2) IN
           LET m1 = compile(e1)(remove(temp, free_e2)) IN
           IF empty?[RT](m1) THEN emptyset[RT]
           ELSE
            LET (code_e1, free_e1) = select(m1) IN
            singleton[RT]((code_e2 ++
                          (: STOREA(temp) :):Code ++
                           code_e1 ++
                          (: BINOPA(bop, temp) :):Code, free))
           ENDIF
          ENDIF
     ENDIF
 ENDCASES
   MEASURE e BY <<
```

A notion of correctness for this compilation is given by predicate `correct_compExpr` in 89. Informally, this predicate states that if the interpretation of the expression code is defined, the value of the expression can be accessed by reading the contents of the accumulator, and the state transition is not vissible on the source state, i.e. locations which are associated with identifiers do not change.

89
```
% === notion of correctness ===

correct_compExpr(e:Expr)(code:Code)  : bool =
 FORALL (start,final:MState):
  interprete(code)(start)(final) IMPLIES
    valmap(ac(final)) = eval(e)(statemap(start)) AND
      statemap(final) = statemap(start)
```

Thus, for proving the correctness of expression compilation in this sense one has to prove:

```
% === correctness of expression compilation ===                          90

expr_compilation_correct : THEOREM
 compile(e)(free)(result)
  IMPLIES correct_compExpr(e)(proj_1(result))
```

The proof is by induction on the structure of expressions. The base cases (constants and identifiers) as well as the induction step for unary operators are proved easily. Here, the most interesting case is the induction step for binary operators. As for the stack machine compilation one first has to prove an invariant in order to accomplish the induction step for binary operators. This invariant states that locations which are not contained in the initial set of temporary locations do not change when executing the code, i.e. only temporaries may change. The proof of the invariant is also by induction on `e`.

Consider now compilation of statements. The specific values provided for the abstract parameters in the generic theory for simple statement compilation consist of:

- The *output* function accesses the accumulator. As in the last subsection, the access location of values is constant, and thus, parameter `T` of the generic theory is not required and instantiated with the unit type.

```
% --- access of target values ---

outputdefd?(u:unit)(ms:MachineState) : bool = true
output(u:unit)(ms:(outputdefd?(u))) : MValue = ac(ms)
```

- Code for storing values into memory at a specific address is given by the single `STORE` instruction:

```
STORE_code(u:unit, a:Addr) : Code = (: STOREA(a):Instr :)
```

- To match the signature of the parameter `compileExpr` we have to change the compilation function for expressions (`compile`) as follows where `t_set` is a fixed set of temporary locations.

```
% --- compilation function of expressions used for instantiation ---

compileExpr(e:Expr) : [(deterministic?[Code]), unit] =
 LET result = compile(e)(t_set) IN
  IF empty?[RT](result) THEN (emptyset[Code], one)
  ELSE LET (code, rest) = choose(result) IN
   (singleton[Code](code), one)
  ENDIF
```

Using these definitions, and let `target_memory` denote the state record selector `mem`, the generic theory can be imported.

```
% --- import compilation of simple statements
 compile_assign[Ident, Expr, SrcValue, eval, Instr, MState,
               one_step, Addr, MValue, unit, outputdefd?,
               RegFile, STORE_code, target_memory,
               representable?, valmap,
               idmap, compileExpr, statemap]
```

Importing this theory, four assumptions are generated, see $\boxed{16}$, $\boxed{18}$, and $\boxed{19}$. Assumption `expression_compilation_correct` is discharged using theorem `expr_compilation_correct` above. Assumption `interprete_store` is proved easily by unfolding definitions. Assumption `symtab_and_memory` is trivial, and finally the proof of assumption `statemap_and_memory` requires injectivity of `idmap`.

Next, we deal with the compilation of control structures. As in the last subsection we do not consider boolean expression compilation explicitly and suppose given a compilation function `compileBExpr` satisfying the assumption `bexp_comp_correct` in $\boxed{91}$. The boolean output function `output_bool` tests the flag in the current state.

```
% === correctness assumption for boolean expression compilation                91

bexp_comp_correct : AXIOM
    (FORALL (b:BExp, c:Code):
      compileBExpr(b)(c) IMPLIES
       FORALL (start,final: MState):
        interprete(c)(start)(final) IMPLIES
            eval_bexp(b)(statemap(start)) = output_bool(final) AND
              statemap(final) = statemap(start))
```

Here, `outputdefd?` is instantiated with the constant true function, and `read` is instantiated with the identity on states. The generic theory for compiling control structures into basic blocks is then imported:

```
% --- import compilation into basic blocks

IMPORTING c2bb[BExp, SState, PId, eval_bexp, SimpleStatement, ss_meaning,
               Instr, MState, one_step, outputdefd?, output_bool,
               read, statemap, compileBExpr, compile_simpleStmt]
```

All generated assumptions are proved in the same way as described in the last subsection. Finally, the generic theory for linearization is imported which enables to prove the main correctness conjecture:

```
% --- compilation of Statements is correct                                    92

stmts_compile_correct : THEOREM
  FORALL (p:source_program): FORALL (start,final:MState):
   compile_defined?(p) AND
     Ip(cp(compile(p)))(start)(final)
      IMPLIES P(p)(statemap(start))(statemap(final))
```

10 Conclusion

In this paper a hierarchy of formal generic theories for the compilation of standard language constructs for procedural languages has been presented. It includes specifications for compiling simple statements, control structures (statements), and parameterless procedures. All specifications are generic in the sense that they abstract from specific target architectures and source languages and can therefore be reused by means of instantiations. The compilation theories are largely independent of each other; they are linked only by parameters and assumptions about compilation functions for constructs lower in the hierarchy. The compilation of control structures, for example, builds on the compilation of expressions and simple statements but can be considered independently. Parameters specify the interface to expression and simple statement compilation for which some correctness assumptions must hold. A further modularization is achieved by splitting the compilation task into small manageable parts following the structure of existing compilers: control structures are first translated into blocks preserving their structure but working on machine states, and then further compiled into linear code by introducing relative jumps. Procedure bodies and the main program are finally linearized. Applicability of the generic theories to specific compilation processes has been demonstrated by means of two examples. All specification and verification tasks have been carried out using the PVS system.

Future work will extend the hierarchy of specifications by new theories for additional language constructs, in particular procedures with parameters, and complex data structures. We are also in the process of instantiating the theories with concrete compilation tasks as defined in the Verifix project, for example, the compilation of the imperative language CINT, a subset of C, into Transputer assembler code. The long term goal of our work is to develop a library of generic specifications which can be utilized to specify and verify different compilation processes of standard imperative and functional languages.

References

[1] F. Bartels, A. Dold, F.W. von Henke, H. Pfeifer, and H. Rueß. Formalizing Fixed-Point Theory in PVS. Ulmer Informatik-Berichte 96-10, Universität Ulm, December 1996.

[2] R.S. Boyer and J S. Moore. A Computer Proof of the Correctness of a Simple Optimizing Compiler for Expressions. Technical Report 5, SRI International, 1977.

[3] M. Broy. Experiences with Software Specification and Verification using LP, the Larch Proof Assistant. Technical report, Digital Systems Research Center, 1992.

[4] Manfred Broy, Ursula Hinkel, Tobias Nipkow, Christian Prehofer, and Birgit Schieder. Interpreter Verification for a Functional Language. In P. S. Thiagarajan, editor, *Proceedings of the 14th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 77–88. Springer-Verlag LNCS 880, 1994.

[5] L.M. Chirica and D.F. Martin. Toward Compiler Implementation Correctness Proofs. *ACM Transactions on Programming Languages and Systems*, 8(2):185–214, April 1986.

[6] Paul Curzon. A Verified Vista Implementation - Final Report. Technical Report 311, University of Cambridge, Computer Laboratory, September 1993.

[7] Wolfgang Goerigk, Axel Dold, Thilo Gaul, Gerhard Goos, Andreas Heberle, Friedrich W. von Henke, Ulrich Hoffmann, Hans Langmaack, Holger Pfeifer, Harald Ruess, and Wolf Zimmermann. Compiler Correctness and Implementation Verification: The *Verifix* Approach. In *CC '96 Int. Conf. on Compiler Construction (poster session)*, Linkøping, Sweden, 1996.

[8] John Hannan and Frank Pfenning. Compiler Verification in LF. In Andre Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, June 1992.

[9] C. A. R. Hoare. Refinement Algebra Proves Correctness of Compiling Specifications. In C.C. Morgan and J.C.P. Woodcock, editors, *3rd Refinement Workshop*, pages 33–48. Springer-Verlag, 1991.

[10] J.J. Joyce. A Verified Compiler for a Verified Microprocessor. Technical Report 167, University of Cambridge, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, England, March 1989.

[11] J S. Moore. A Mechanically Verified Language Implementation. *Journal of Automated Reasoning*, 5(4), 1989.

[12] J S. Moore. *Piton, A Mechanically Verified Assembly-Level Language.* Kluwer Academic Publishers, 1996.

[13] Markus Müller-Olm. An Exercise in Compiler Verification. Internal report, CS Department, University of Kiel, 1995.

[14] Markus Müller-Olm. *Modular Compiler Verification.* PhD thesis, Techn. Fakultät der Christian-Albrechts-Universität, Kiel, June 1996.

[15] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[16] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *Proceedings 11th International Conference on Automated Deduction CADE*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, October 1992. Springer-Verlag.

[17] H. Pfeifer, A. Dold, F.W. von Henke, and H. Rueß. Mechanized Semantics of Simple Imperative Programming Constructs. Ulmer Informatik-Berichte 96-11, Universität Ulm, December 1996.

[18] W. Polak. Compiler Specification and Verification. In J. Hartmanis G. Goos, editor, *Lecture Notes in Computer Science*, number 124 in LNCS. Springer-Verlag, 1981.

[19] Augusto Sampaio. A Comparative Study of Theorem Provers: Proving Correctness of Compiling Specifications. Technical report, Oxford University Computing Laboratory, Programming Research Group, 1991.

[20] Augusto Sampaio. *An Algebraic Approach to Compiler Design.* PhD thesis, Oxford University Computing Laboratory, Programming Research Group, October 1993. Technical Monograph PRG–110, Oxford University Computing Laboratory.

[21] Deborah Weber-Wulff. Proof Movie – A Proof with the Boyer-Moore Prover. *Formal Aspects of Computing*, 5(2):121–151, 1993.

[22] Phillip J. Windley. A Theory of Generic Interpreters. In George J. Milne and Laurence Pierre, editors, *Correct Hardware Design and Verification Methods*, volume 683 of *Lecture Notes in Computer Science*, pages 122–134. Springer-Verlag, May 1993.

[23] William D. Young. A Mechanically Verified Code Generator. *Journal of Automated Reasoning*, 5:493–518, 1989.