Appeared as technical report UIB-96-11, Universität Ulm, Fakultät für Informatik, Dec. 1996

Mechanized Semantics of Simple Imperative Programming Constructs*

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

Abt. Künstliche Intelligenz
Fakultät für Informatik
Universität Ulm
D-89069 Ulm

verifix@ki.informatik.uni-ulm.de

Abstract

In this paper a uniform formalization in PVS of various kinds of semantics of imperative programming language constructs is presented. Based on a comprehensive development of fixed point theory, the denotational semantics of elementary constructs of imperative programming languages are defined as state transformers. These state transformers induce corresponding predicate transformers, providing a means to formally derive both a weakest liberal precondition semantics and an axiomatic semantics in the style of Hoare. Moreover, algebraic laws as used in refinement calculus proofs are validated at the level of predicate transformers. Simple reformulations of the state transformer semantics yield both a continuation-style semantics and rules similar to those used in Structural Operational Semantics.

This formalization provides the foundations on which formal specification of programming languages and mechanical verification of compilation steps are carried out within the *Verifix* project.

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

Contents

1. Introduction 1

1.1 Related Work . . . 1

1.2 Overview . . . 2

2. A brief description of PVS 4

3. Mechanizing Domain Theory 5

4. Denotational Semantics for Statements 6

4.1 State Transformers . . . 6

4.2 State Transformer Semantics . . . 8

4.3 Predicate Transformers . . . 13

5. Relating Various Styles of Semantics 15

5.1 Weakest (Liberal) Precondition . . . 15

5.2 Hoare Calculus . . . 16

5.3 Algebraic Laws . . . 17

5.4 Structural Operational Semantics . . . 19

5.5 Continuations . . . 21

6. Introducing Variable Declarations 21

7. Example: A Simple Programming Language 24

8. Conclusions 25

Appendix 29

A. Denotational Semantics 31

A.1 State transformers . . . 31

A.2 Program State . . . 31

A.3 State Transformer Semantics . . . 34

A.4 Predicates . . . 37

A.5 Predicate Transformers . . . 39

A.6 Predicate Transformer Semantics . . . 39

B. Various Styles of Semantics **41**

B.1 Weakest Precondition 41
B.2 Hoare Logic 42
B.3 Algebraic Laws 43
B.4 SOS 45
B.5 Continuation 46
B.6 Putting it Together 47

C. A Simple Programming Language **47**

C.1 Simple Commands 47
C.2 Simple Declarations 48
C.3 The Language 48

D. Proof of Lemma `while_strans_adm` **49**

1. Introduction

Formal verification of compiler implementations requires the semantics of the programming languages involved be modeled adequately in a formal system. Depending on the aspect of a programming language or its compilation one wants to describe, certain styles of semantics are more suitable than others. A mathematically precise definition of a language, for example, would often be accomplished using denotational semantics [SS72, Ten76, Sto77]. On the other hand, if one is interested in developing a prototype implementation one might describe the semantics of the language using an operational style, while an axiomatics semantics [Flo67, Hoa69] would be most useful if verification of programs is of concern.

In this paper we present a generic modelling in PVS of different kinds of semantics of the basic imperative programming constructs. Starting from a denotational semantics based on state transformers we show how to derive the characteristic rules of various semantic styles including weakest precondition semantics, Hoare logic, continuation-style semantics, and structural operational semantics. The formalization is modular in the sense that there are separate PVS specifications for each programming construct.

In addition, the components of a programming language such as expressions, statements, and variable declarations are identified with their semantics. Thus, only the abstract structure of the language's components is laid down and the formalization can hence be applied to every programming language that consists of one or more of the constructs covered in this paper.

The mathematical theory underlying the work presented here is well understood. This paper adds nothing new in this respect; in fact, most of the theorems can be found in standard textbooks, such as [Win93, Sch88, Bes95]. However, we think that the main contribution of our work is a uniform treatment of different styles of semantics in a formal system. This allows for doing formal reasoning about both programs and their compilation within the same framework. Moreover, since many interchangeable forms of semantics are available, the reasoning can be carried out selecting that semantic formalism suiting best.

All theorems presented here have been proved within PVS. Most of the theories are reprinted in the Appendix. The complete specifications and proofs are available from the authors upon request.

1.1 Related Work

There have been several reports related to the modelling of semantics of programming languages in a theorem prover, including [Sok87, Mas87, Cam90, HM95], and there are standard techniques for formalizing various styles of semantics. For example, our presentation regarding predicate transformers and algebraic semantics is comparable to a formalization in the HOL system of a guarded command language with weakest precondition semantics by Back and v. Wright [BvW90].

The work presented here is most closely related to that of Gordon [Gor89] and Nipkow [Nip96], since in contrast to the papers mentioned above they deal with several forms of semantics. Gordon uses the HOL system to mechanically derive the rules of Hoare logic of

a simple while-language from their denotational semantics [Gor89]. Furthermore, he describes how to apply HOL tactics to generate verification conditions based on the derived Hoare rules. Nipkow [Nip96] presents a formalization of the first chapters of Winskel's textbook on programming language semantics [Win93] in the theorem prover Isabelle/HOL. He defines operational, denotational, and axiomatic semantics of a simple imperative programming language and proves their equivalence. Additionally, proofs of soundness and completeness of a verification condition generator are sketched.

In contrast to Nipkow, we do not define different forms of semantics for the same language, but follow Gordon and show how to derive the characteristic rules of the various semantic formalisms from a denotational semantics. This saves us from having to prove explicitly the equivalence of the different semantics.

1.2 Overview

The remainder of the paper is organized as follows: after giving a brief overview of the specification and verification system PVS in Section 2, the remaining sections describe the various PVS theories developed. The overall structure of these theories is illustrated in Figure 1. Each node in this hierarchy graph represents one PVS theory, and the edges represent importing relations. Theories towards the bottom are used (imported) by those above them.

In order to be able to define the denotational semantics of loops and recursive procedures we have developed a rather comprehensive formalization of the theory of fixed points in PVS. This formalization is only briefly summarized in Section 3; a detailed description can be found in a companion paper [BDvH$^+$96]. Consequently, the bottom entry in Figure 1 is to be regarded as standing for a whole cluster of theories.

Climbing up the hierarchy, Section 4.1 describes the concept of state transformers that are defined in theory srel. Next, the denotational semantics of imperative programming language constructs based on state transformers is presented in Section 4.2. The definitions are split into several theories and are combined in theory statements. Predicate transformers as defined in the theories ptrans and mpt are introduced in Section 4.3. State transformers and predicate transformers provide the basis for a uniform treatment of various styles of semantics such as weakest precondition semantics, Hoare-style semantics, and algebraic laws, see Section 5. The corresponding theories can be found further up in the theory hierarchy and are comprised by the theory semantics.

The definitions in the theories mentioned so far abstract from the state on which programs operate. A concrete implementation of such a state is given in Section 6. Finally, as an example, Section 7 describes how the formalizations can be used to define the semantics of a simple while language.

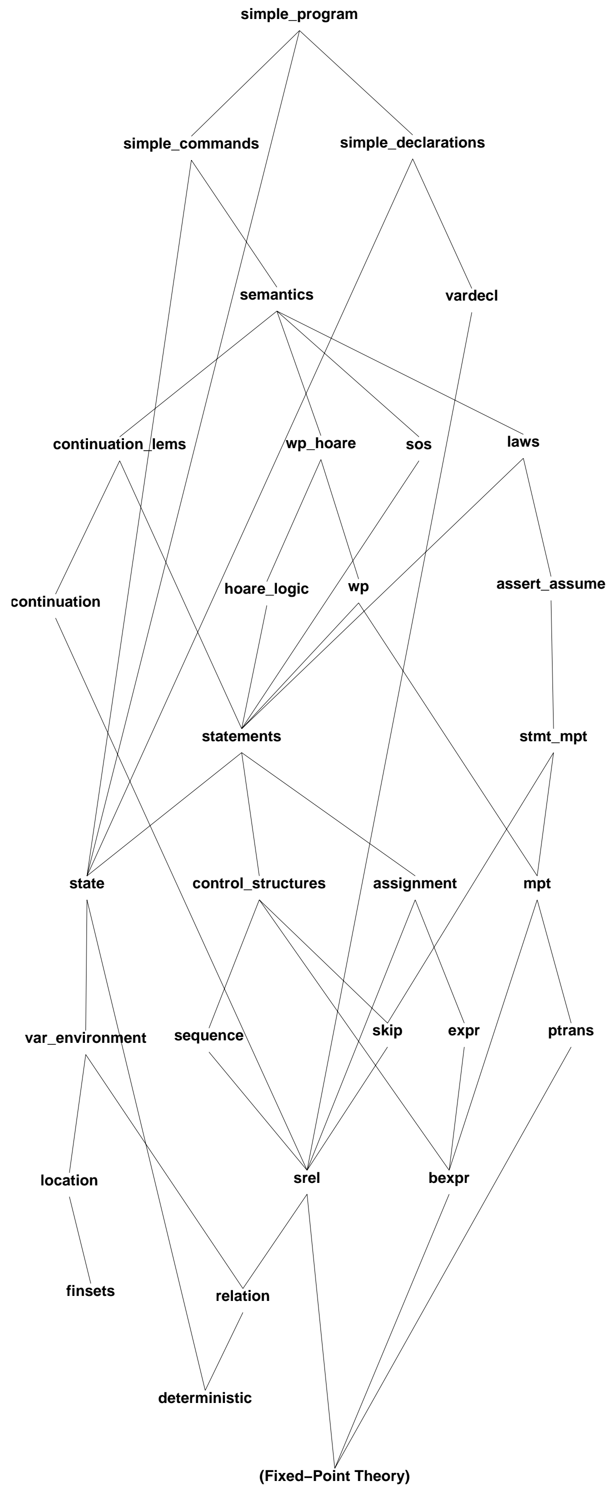

Figure 1: Hierarchy of PVS theories formalizing various kinds of semantics of simple constructs in imperative programming languages.

2. A brief description of PVS

This section gives a brief overview of PVS. More details can be found in [ORSvH95].

The PVS system combines an expressive specification language with an interactive proof checker that has a reasonable amount of theorem proving capabilities. It has been used for reasoning in domains as diverse as microprocessor verification, protocol verification, and algorithms and architectures concerning fault-tolerance [ORSvH95].

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.

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. If an expression that produces a TCC is used many times, the typechecker repeatedly generates the same TCC. The use of *judgements* can prevent this. There are two kinds of judgements:

```
JUDGEMENT +          HAS_TYPE   [even, even -> even]
JUDGEMENT continuous SUPTYPE_OF monotonic
```

The first form, a constant judgement, asserts a closure property of `+` on the subtype of even natural numbers. The second one, a subtype judgement, asserts that a given type is a subtype of another type. The typechecker generates a TCC for each judgement to check the validity of the assertion, but will then use the information provided further on. Thus, many TCCs can be suppressed.

PVS specifications are packaged as *theories* that can be parametric in types and constants. A built-in *prelude* and loadable *libraries* provide standard specifications and proved facts for a large number of theories.

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.

3. Mechanizing Domain Theory

In order to be able to define the denotational semantics of loops and recursive procedures we have developed a rather comprehensive formalization of the theory of fixed points in PVS. In this section, we only summarize the main concepts needed in this paper. A much more detailed description can be found in [BDvH+96].

The formalization consists of a collection of PVS theories that can be used as a library. The most important theorems are the well-known schemes of fixed-point induction on monotonic and continuous functions.[1]

```
                                                                    1
fp_induction_mono : THEOREM
  FORALL (P:(admissible?), f:(monotonic?)):
    P(bottom) AND (FORALL (x:D): P(x) IMPLIES P(f(x)))
      IMPLIES P(mu(f))

fp_induction_cont : THEOREM
  FORALL (P:(admissible?), g:(continuous?)):
    P(bottom) AND
      (FORALL (i:nat): P(iterate(g,i)(bottom))
        IMPLIES P(iterate(g,i + 1)(bottom)))
    IMPLIES P(mu(g))
```

Throughout this section we use `f` and `g` to denote a monotonic and a continuous function, respectively, over some *complete partial order* `D`. The least element of `D` with respect to the ordering relation `<=` is called `bottom`. We use the term *pre-cpo* for cpo's without such a least element. For convenience, we repeat the definitions concerning cpo's (let `b` be of type `D` and `<=` a partial order on `D`):

```
                                                                    2
bottom?(<=)(b) : bool =
  FORALL (x:D): b <= x

precpo?(<=) : bool = FORALL (C: Chain[D,<=]): lub_exists?(C)
pCPO: TYPE = (precpo?)

cpo?(<=,b) : bool =
  precpo?(<=) AND bottom?(<=)(b)
CPO : TYPE = (cpo?)
```

Sometimes we use a special kind of pre-cpo's where the ordering relation is simply the identity. Such pre-cpo's are referred to as *discrete pre-cpo's*. Note that every function from a discrete pre-cpo into a pre-cpo is continuous.

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. Thus, when typechecking expressions such as `mu(f)`, PVS generates a corresponding type correctness condition.

[1] PVS notation: For predicates `P` over some type `A`, the expression `(P)` is an abbreviation for the predicate subtype `{x:A | P(x)}`.

```
% f : VAR (monotonic?)        % this is a comment!                      3
% g : VAR (continuous?)

least_fixpoint?(f) : set[D] =
  {x:D | fixpoint?(x,f) AND FORALL (y:D): fixpoint?(y,f) IMPLIES x <= y}

LFP(f) : TYPE = (least_fixpoint?(f))

mu_exists?(f) : bool = nonempty?(least_fixpoint?(f))

mu(f:(mu_exists?)) : LFP(f) =
  choose(least_fixpoint?(f))

mu_is_fixpoint : LEMMA
  FORALL (f:(mu_exists?)): f(mu(f)) = mu(f)
```

Monotonic functions mapping into a cpo always have least fixpoints – this is known as the *Knaster-Tarski Theorem*. In the case of continuous functions, the least fixed point can be calculated as the least upper bound of all i^{th} iterations of the function applied to the bottom element of the cpo.

```
Knaster_Tarski : THEOREM                                                  4
  mu_exists?(f)

fixpoint_theorem : THEOREM
  mu(g) = lub({x | EXISTS (i:nat): x = iterate(g,i)(bottom)})
```

4. Denotational Semantics for Statements

4.1 State Transformers

The notion of state transformers provides the basis for the denotational semantics of statements. State transformers are defined in theory srel. In order to be able to deal with nontermination we use relations rather than functions; the latter have to be total in PVS. A relation $R \subseteq A \times B$ is modeled as a function mapping elements of type A to a set of elements of type B. Partial functions can also be described by restricting the range to sets with at most one element.

```
Relation : TYPE = [A -> set[B]]                                           5

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

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

Let `R` be a relation, `S` a set of elements of type `A`, and `T` a set over type `B`. We define the *image* of `S` under `R` and the *inverse image* of `T` under `R` by:[2]

[2] In PVS sets are identified with their characteristic predicates and thus the expressions `pred[sigma]` and `set[sigma]` are interchangeable. Hence, the notation `s:(S)` also means that `s` is an element of set `S`.

```
image(R,S)         : set[B] = {y:B | EXISTS (s:(S)): member(y,R(s))}     6
inverse_image(R,T) : set[A] = {x:A | subset?(R(x),T)}
```

In order to keep the definition of state transformers as general as possible, the state on which programs operate is left uninterpreted here. Instead, uninterpreted types `sigma` and `tau`, provided as theory parameters, are used for representing the domain and range types of state transformers, respectively. The parameters are usually instantiated with a concrete implementation of the program state, such as the one described in Section 6.

In the most general case, nondeterministic state transformers are relations between `sigma` and `tau`. We use the type `srel` for such relations, while deterministic state transformers have type `strans`, obviously[3] a subtype of `srel`.

```
srel   : TYPE = Relation[sigma,tau]                                      7

strans : TYPE = PartialFunction[sigma,tau]
```

A partial ordering `<=` on `srel` is defined by pointwise lifting set inclusion. As this mechanism is often used, a small theory pointwise[4] is defined. It is parameterized with two types `D` and `R`, and a partial ordering `leq` on `R`. The theory defines a new partial ordering on the function space `[D -> R]` by a pointwise extension of `leq`.

```
pointwise[D, R : TYPE+,                                                  8
          leq : (partial_order?[R])] : THEORY
BEGIN

  f,g : VAR [D -> R]

  <=(f,g) : bool = FORALL (x:D): leq(f(x), g(x))

  JUDGEMENT <= HAS_TYPE (partial_order?[[D -> R]])

END pointwise
```

The partial ordering on `srel` is obtained by importing the theory pointwise. The parameters are instantiated with the domain and range of `srel`. The partial ordering on the range of `srel` is set inclusion, which is itself defined by instantiating pointwise appropriately.

```
IMPORTING pointwise[tau,bool,=>]                                         9
IMPORTING pointwise[sigma,set[tau],pointwise[tau,bool,=>].<=]
```

The state transformer mapping every state to the empty set is the least element with respect to `<=` and is called `abort`. Since every type `sigma` forms a discrete cpo and sets together with set inclusion are a cpo, functions of type `srel` are continuous. Moreover, `srel` and `<=` form a cpo with bottom element `abort`.

[3] It is even obvious to the typechecker: a corresponding judgement `JUDGEMENT strans SUBTYPE_OF srel` is rejected as it does not provide any additional information.

[4] The theory pointwise is contained in the library for fixed-point theory.

```
abort : srel =                                                          10
  LAMBDA (s:sigma): emptyset

JUDGEMENT <=    HAS_TYPE pCPO[srel]
JUDGEMENT abort HAS_TYPE (bottom?(<=))
```

4.2 State Transformer Semantics

This section describes the formalization of a denotational semantics based on state transformers for basic statements of imperative programming languages. The definitions are split into several theories, one for each construct. The theories are parameterized with respect to some type `sigma`, which is intended to model the state on which programs operate.

Since it does not add anything interesting, we do not distinguish different kinds of expressions. Instead, expressions are identified with their semantics and are modeled as a valuation function.

```
expr[sigma, Value: TYPE+] : THEORY                                      11

BEGIN

  Expr : TYPE = [sigma -> Value]  % --- modeled as semantic function

END expr
```

In addition, Boolean expressions `BExpr` are defined as predicates on `sigma`.

```
BExpr : TYPE = pred[sigma]                                              12
```

Primitive Statement: `skip`

The simplest statement is `skip`, which does not change the current state and hence is modeled as the function mapping a state s to the singleton set $\{s\}$:

```
skip [sigma : TYPE+] : THEORY                                           13
BEGIN

  IMPORTING srel[sigma,sigma]

  skip : srel =
    LAMBDA (s:sigma): singleton(s)

  JUDGEMENT skip HAS_TYPE strans

END skip
```

Sequential Composition

Sequential composition of two state transformers `f` and `g`, denoted by `f ++ g`, is defined by relational composition.

```
++(f, g) : srel =                                                    14
  LAMBDA s: image(g,f(s))

sequence_strans_closed : LEMMA
  FORALL (f,g:strans): deterministic?((f ++ g)(s))

JUDGEMENT ++ HAS_TYPE [strans, strans -> strans]
```

Sequential composition is monotonic in both arguments.

```
sequence_monotonic_left : LEMMA                                      15
  f <= g IMPLIES f ++ h <= g ++ h

sequence_monotonic_right : LEMMA
  f <= g IMPLIES h ++ f <= h ++ g
```

Assignment

The theory defining the semantics of assignment statements has some additional parameters: the type of variables and values, a predicate for deciding whether or not a variable is declared in the current state, and a function `update` for assigning a new value to a previously declared variable.

```
assignment [Vars, Value, sigma : TYPE+,                              16
            declared? : [sigma -> pred[Vars]],
            update    : [s:sigma -> [(declared?(s)), Value -> sigma]]
           ] : THEORY
```

The semantics of assigning the value of an expression `e` to some variable `x`, denoted by `x << e`, is a new state with `x` being bound to `e`, if `x` is declared. Otherwise, the assignment statement is undefined.

```
IMPORTING expr[sigma, Value]                                         17

s : VAR sigma; x : VAR Vars; e : VAR Expr

<<(x,e) : srel =                  % --- prefix notation of infix operator <<
  LAMBDA s: IF declared?(s)(x)
              THEN singleton(update(s)(x,e(s)))
              ELSE emptyset
            ENDIF

JUDGEMENT << HAS_TYPE [Vars,Expr -> strans]
```

Conditional Branching and Loops

The semantic function for the conditional `IF-THEN-ELSE` is obtained by lifting the boolean `IF`-expression.

```
IF(b:BExpr, f, g:srel) : srel =                                      18
  LAMBDA s: IF b(s) THEN f(s) ELSE g(s) ENDIF;
```

Again, we have monotonicity in both recursive arguments, and IF is deterministic provided the branches are.

```
                                                                    19
IF_monotonic_left : LEMMA
  f <= g IMPLIES
   IF b THEN f ELSE h ENDIF  <=  IF b THEN g ELSE h ENDIF

IF_monotonic_right : LEMMA
  f <= g IMPLIES
   IF b THEN h ELSE f ENDIF  <=  IF b THEN h ELSE g ENDIF

IF_strans_closed : LEMMA
  FORALL (f,g:strans): deterministic?((IF b THEN f ELSE g ENDIF)(s))
```

The semantics of the `while` loop is defined as usual as the least fixed point[5] of a functional describing one iteration of the `while`-loop. As the domain of `mu` is defined by a predicate subtype, see 3, typechecking the definition of `while` yields a type correctness condition. One has to prove that the above functional does have a least fixpoint, which is done using monotonicity of `IF` and `++`, and the Knaster-Tarski theorem (4).

```
                                                                    20
while_functional_monotonic : LEMMA
  monotonic?(LAMBDA (x:srel): IF b THEN f ++ x ELSE skip ENDIF)

while(b, f) : srel =
    mu! (x:srel): IF b THEN f ++ x ELSE skip ENDIF
```

As an example, we step through the proof of the lemma `while_strans_closed`. It states that the `while` loop is deterministic, provided the body is. It is used for proving the judgement following right after.

```
                                                                    21
while_strans_adm : LEMMA admissible?(LAMBDA (y: srel): FORALL (s: sigma):
  deterministic?(y(s)))

while_strans_closed : LEMMA
  FORALL (f:strans): FORALL (s:sigma): deterministic?(while(b,f)(s))

JUDGEMENT while HAS_TYPE [[BExpr, strans] -> strans]
```

For presentation purposes we slightly edit the PVS output, e. g. by deleting unnecessary formulae.

```
while_strans_closed :

  |-------
{1}    FORALL (b: BExpr), (f: strans):
        FORALL (s: sigma): deterministic?(while(b, f)(s))
```

The proposition is proved using fixed-point induction. In order to prepare the formula for the application of the induction theorem, some of the quantifiers are eliminated, followed by unfolding the definition of `while`.

[5] The notation `mu! (x:T): f(x)` is an abbreviation for `mu(LAMBDA (x:T): f(x))`.

```
Rule? (THEN (SKOLEM!) (EXPAND "while"))

while_strans_closed :

  |-------
{1}    FORALL (s: sigma):
        deterministic?(mu! (x: srel):
                          IF b!1 THEN f!1 ++ x ELSE skip ENDIF(s))
```

Now the theorem of fixed-point induction for monotonic functions (see [1]) is applied with the predicate P instantiated appropriately. This proof step yields three subgoals. The first one is the induction rule itself, whereas the other ones (see below) are type correctness conditions. These are generated since the fixed-point induction theorem is constrained to admissible predicates P and monotonic functions f.

```
Rule? (USE "fp_induction_mono" :SUBST
         ("P" "LAMBDA (y:srel): FORALL (s:sigma): deterministic?(y(s))"))

while_strans_closed.1 :

{-1}    (FORALL (s: sigma): deterministic?(abort(s))
          AND
         (FORALL (x: srel):
           FORALL (s: sigma): deterministic?(x(s)) IMPLIES
            FORALL (s: sigma):
             deterministic?((IF b!1 THEN f!1 ++ x ELSE skip ENDIF)(s))))
        IMPLIES
          FORALL (s: sigma):
            deterministic?(mu(LAMBDA (x: srel):
                               IF b!1 THEN f!1 ++ x ELSE skip ENDIF)(s))
  |-------
[1]    FORALL (s: sigma):
        deterministic?(mu! (x: srel):
                          IF b!1 THEN f!1 ++ x ELSE skip ENDIF(s))
```

Propositional simplification of the first subgoal yields another two subgoals corresponding to the induction base and the induction step.

```
Rule? (PROP)

while_strans_closed.1.1 :

  |-------
{1}    FORALL (s: sigma): deterministic?(abort(s))
```

The induction base is simply proved by unfolding definitions and propositional reasoning.

```
Rule? (GRIND)

This completes the proof of while_strans_closed.1.1.

while_strans_closed.1.2 :

  |-------
{1}    FORALL (x: srel):
         FORALL (s: sigma): deterministic?(x(s))
```

```
IMPLIES FORALL (s: sigma):
  deterministic?(IF b!1 THEN f!1 ++ x ELSE skip ENDIF(s))
```

The proof of the induction step is accomplished using the corresponding properties of the defining expressions of `while`, that is `IF` and the sequencing constructor `++`. Conditionals are deterministic, given the branches are. However, using the lemma `IF_strans_closed` yields only one subgoal for the `THEN` branch. This is due to the judgement in 13: the typechecker already knows that `skip` is of type `strans` and therefore suppresses this subgoal.

```
Rule? (THEN (SKOSIMP*) (REWRITE "IF_strans_closed"))

while_strans_closed.1.2 :

[-1]    FORALL (s: sigma): deterministic?(x!1(s))
  |-------
{1}    FORALL (s: sigma): deterministic?[sigma]((f!1 ++ x!1)(s))
```

Next, lemma `sequence_strans_closed` is applied. Again, only one subgoal is generated since by definition `f!1` is of type `strans`.

```
Rule? (SKOSIMP*) (REWRITE "sequence_strans_closed")

while_strans_closed.1.2 :

[-1]    FORALL (s: sigma): deterministic?(x!1(s))
  |-------
{1}    FORALL (s: sigma): deterministic?(x!1(s))
```

Finally, the induction hypothesis is used to complete this part of the proof.

```
Rule? (SKOSIMP*) (INST?)

This completes the proof of while_strans_closed.1.2.

This completes the proof of while_strans_closed.1.

while_strans_closed.2 (TCC):

  |-------
{1}    monotonic?(LAMBDA (x: srel): IF b!1 THEN f!1 ++ x ELSE skip ENDIF)
```

We have still to prove that the application of the fixed-point induction theorem is valid. First, monotonicity of the defining functional of `while` has to be shown. As this is also required in the definition of `while`, a seperate lemma has been established (20), which is itself proved using monotonicity of `IF` and `++`, and the Knaster-Tarski theorem.

```
Rule? (REWRITE "while_functional_monotonic")

This completes the proof of while_strans_closed.2.

while_strans_closed.3 (TCC):

  |-------
{1}    admissible?(LAMBDA (y: srel):
                     FORALL (s: sigma): deterministic?(y(s)))
```

The proof that the predicate used in the fixed-point induction is admissible is rather technical. At this point, we simply use an appropriate lemma and refer to Appendix D for a complete proof script.

```
Rule? (USE "while_strans_adm")

This completes the proof of while_strans_closed.3.

Q.E.D.
```

Finally, two useful lemmata are provided. The first one, `while_def`, describes the unfolding of a `while` loop, the second one states that `while` is monotonic in its recursive argument.

```
while_def : LEMMA                                                          22
  while(b,f) = (IF b THEN f ++ while(b,f) ELSE skip ENDIF)

while_montonic : LEMMA
  f <= g IMPLIES while(b,f) <= while(b,g)
```

The proofs are by rewriting theorems provided by the fixed-point theory, such as the Knaster-Tarski theorem or Park's Lemma.

4.3 Predicate Transformers

This section briefly describes the formalization of predicate transformers. In PVS, predicates over some type `sigma` are, as usual, boolean-valued functions on `sigma`. Some simple predicates are obtained by lifting the standard boolean connectives, see theory predicates.

```
TRUE      : pred[sigma] = LAMBDA s: TRUE;                                  23
FALSE     : pred[sigma] = LAMBDA s: FALSE;
NOT(p)    : pred[sigma] = LAMBDA s: NOT(p(s));
/\(p,q)   : pred[sigma] = LAMBDA s: p(s) AND q(s);
\/(p,q)   : pred[sigma] = LAMBDA s: p(s) OR q(s);
=>(p,q)   : pred[sigma] = LAMBDA s: p(s) IMPLIES q(s);
<=>(p,q)  : pred[sigma] = LAMBDA s: p(s) IFF q(s);
```

Implication on boolean values is extended pointwise to obtain a partial ordering `<=` on predicates. Since every type `sigma` is a discrete cpo and the type of boolean expressions `bool` together with implication is a cpo, we have (`pred[sigma]`,`<=`) being a cpo. As predicates and sets are interchangeable, `<=` can also be interpreted as set inclusion.

```
% -------- Ordering on predicates:                                         24
%          the next IMPORT defines a partial ordering <= on predicates as
%          p <= q  :<=>  FORALL s: p(s) IMPLIES q(s)

IMPORTING pointwise[sigma,bool,=>]

JUDGEMENT <= HAS_TYPE pCPO[pred[sigma]]
```

In theory ptrans, predicate transformers are defined as functions with domain pred[tau] and range pred[sigma] for some types sigma and tau. Again, we use a lifting mechanism to define basic predicate transformers and a partial ordering <=.

```
                                                                        25
ptrans : TYPE = [pred[tau] -> pred[sigma]]

abort   : ptrans = LAMBDA q: FALSE
magic   : ptrans = LAMBDA q: TRUE
NOT(F)  : ptrans = LAMBDA q: NOT(F(q))
/\(F,G) : ptrans = LAMBDA q: F(q) /\ G(q)
\/(F,G) : ptrans = LAMBDA q: F(q) \/ G(q)
=>(F,G) : ptrans = LAMBDA q: F(q) => G(q)

% -------- Ordering
%          next IMPORT defines partial ordering <= on predicates transformers
%          as F <= G  :<=>  FORALL q: F(q) <= G(q)

IMPORTING pointwise[set[tau],set[sigma],pointwise[sigma,bool,=>].<=]

JUDGEMENT <=    HAS_TYPE pCPO[ptrans]
JUDGEMENT abort HAS_TYPE (bottom?(<=))
```

We generalize conjunction and disjunction of predicate transformers and define operators for greatest lower bounds and least upper bounds of sets of predicate transformers, that is, meets and joins.

```
                                                                        26
M : VAR set[ptrans]
/\(M): ptrans = LAMBDA q: LAMBDA y: FORALL F: M(F) IMPLIES F(q)(y)
\/(M): ptrans = LAMBDA q: LAMBDA y: EXISTS F: M(F) AND F(q)(y)
```

The type MPT constrains predicate transformers to monotonic predicate transformers.

```
                                                                        27
MPT : TYPE = {F:ptrans | monotonic?(F)}

JUDGEMENT abort HAS_TYPE MPT
JUDGEMENT magic HAS_TYPE MPT
JUDGEMENT /\    HAS_TYPE [MPT,MPT -> MPT]
JUDGEMENT \/    HAS_TYPE [MPT,MPT -> MPT]
```

Next, we relate state transformers and monotonic predicate transformers by defining an operator mapping every state transformer to a corresponding predicate transformer, cf. theory mpt. Both state and predicate transformations can be viewed as being applied in one of two directions: "forward" transformations map initial states and pre-conditions to final states and post-conditions, respectively, while "backward" transformations do it the other way round. Since our goal is to define a weakest *pre*-condition semantics we adopt the view of applying predicate transformers backwards, while thinking of state transformer as operating forwards.

The operator PT takes a state transformer f of type [sigma -> set[tau]] and yields a predicate transformer of type [pred[tau] -> pred[sigma]]. Given a post-condition T, the expression PT(f)(T) denotes the set of all states s of type sigma, such that all states in f(s) satisfy T.

```
PT(f:srel) : ptrans =                                              28
  LAMBDA (T:pred[tau]): inverse_image(f,T)
```

The operator PT is shown to be universally conjunctive and to map state transformers to monotonic predicate transformers. Moreover, lemma lifting describes the relationship between state transformers and predicate transformers.

```
JUDGEMENT PT HAS_TYPE [srel -> MPT]                                29

PT_universally_conjunctive : LEMMA
  FORALL (M:set[pred[tau]]):
    PT(f)(/\(M)) =
      /\({p:pred[sigma] | EXISTS (m:(M)): p = PT(f)(m)})

lifting : LEMMA
  f <= g IFF PT(g) <= PT(f)
```

5. Relating Various Styles of Semantics

After having formalized the foundations for denotational semantics of imperative programming language constructs, we can now develop other styles of semantics based on the state transformer and predicate transformer styles. In this section, a weakest (liberal) precondition semantics and Hoare-style semantics are developed and shown to be equivalent. Moreover, we describe how algebraic laws, as utilized e. g. in refinement calculus proofs, are validated. Last, we briefly sketch the relationship between state transformer semantics and both structural operational semantics and continuation semantics.

5.1 Weakest (Liberal) Precondition

The operator PT defined in section 4.3, see 28, obviously maps a state transformer f to a predicate transformer F that, applied to some predicate q, yields the *weakest (liberal) precondition* of f with respect to q.

```
wp(f,q) : pred[State] = PT(f)(q)                                   30
```

In order to demonstrate the correctness of this definition, some basic facts about wp are proved. First, wp(f,q) is indeed the *weakest* precondition.

```
wp_char : LEMMA                                                    31
  wp(f,q) = \/({p | p <= PT(f)(q)})
```

The well-known rules of weakest preconditions of basic imperative programming language constructs can easily be proved from the definition of wp.[6]

[6] Function subst is defined in theory state, see Appendix A.2.

32
```
wp_skip : LEMMA
  wp(skip, q) = q

wp_assign : LEMMA
  wp(x << e, q) = subst(q, x, e)

wp_if : LEMMA
  wp(IF b THEN f ELSE g ENDIF, q)
    = (IF b THEN wp(f, q) ELSE wp(g, q) ENDIF)

wp_seq : LEMMA
  wp(f ++ g, q) = wp(f,wp(g,q))

wp_while : LEMMA
  wp(while(b, h), q)
    = (IF b THEN wp(h ++ while(b, h),q) ELSE q ENDIF)
```

5.2 Hoare Calculus

This subsection introduces the definition of Hoare triples and shows how the rules and axioms of Hoare Logic are derived from the denotational semantics definition. We use the notation `|=(p,f,q)` to denote the Hoare-triple $\{p\}f\{q\}$. Its definition is just a translation of the informal description into PVS syntax: {`p`}`f`{`q`} holds, if whenever `f` is executed in a state satisfying `p`, and if the execution terminates, then `f` yields a state where `q` holds. That is, a program segment `f` must map the *set* `p` into `q`.

33
```
|=(p, f, q) : bool = image(f,p) <= q
```

The well-known rules for statements are proved easily. In fact, PVS can do most of the proofs with a single strategy called `(GRIND)`, that repeatedly rewrites definitions, lifts `IF`-expressions and simplifies with decision procedures.

34
```
skip_rule : LEMMA
  |=(p, skip, p)

assign_rule : LEMMA
  |=(subst(p, x, e), (x << e), p)

seq_rule : LEMMA
    |=(p, f, q) AND |=(q, g, r)
  IMPLIES
    |=(p, f ++ g, r)

if_rule : LEMMA
    |=(p /\ b, f, q) AND |=(p /\ NOT(b), g, q)
  IMPLIES
    |=(p, (IF b THEN f ELSE g ENDIF), q)

while_rule : LEMMA
    |=(p /\ b, h, p)
  IMPLIES
    |=(p, while(b, h), p /\ NOT(b))
```

Moreover, useful rules concerning strengthening respectively weakening are proved.

35

```
precondition_strengthening : LEMMA
    p <= p1 AND |=(p1, f, q)
  IMPLIES
    |=(p, f, q)

postcondition_weakening : LEMMA
    |=(p, f, q1) AND q1 <= q
  IMPLIES
    |=(p, f, q)

rule_of_consequence : COROLLARY   % --- combination of the above
    p <= p1 AND |=(p1, f, q1) AND q1 <= q
  IMPLIES
    |=(p, f, q)
```

Equivalence of Hoare logic and weakest precondition semantics can also be proved quite easily using (GRIND).

36

```
hoare_logic_equiv_wp : THEOREM
  |=(p, f, q) = (p <= wp(f, q))
```

5.3 Algebraic Laws

Back [Bac80, Bac81] introduced the *refinement calculus* as a formalization of the stepwise refinement approach to systematic program construction [Dij71, Wir71]. A program statement S' is said to refine S, denoted $S \leq S'$, if and only if S' satisfies every assertion that S does. Program refinements $S \leq S'$ are often proved using several high-level refinement rules that express certain *laws* of the programming language, cf. [HHJ$^+$87]. These laws are justified using monotonic predicate transformer semantics [Bac89].

The formalizations presented so far are sufficient to prove some typical laws. In fact, the definition of program refinement as stated above has already been introduced, cf. lemma `lifting` in specification 29 and the definition of `PT` in 28.

37

```
refinement_def : LEMMA
  f <= g IFF FORALL (q:pred[sigma]): wp(g, q) <= wp(f, q)
```

Some of the laws following next can be proved at the level of state transformers, while others have to be lifted to the domain of predicate transformers. However, PVS provides a useful mechanism of overloading together with implicitly applied (type) conversions, such that this difference can be hidden in the specifications. In order to exploit this feature, we define additional predicate transformers encoding sequential and conditional composition, see theory stmt_mpt.

38

```
F,G : VAR ptrans

++(F,G)   : ptrans = F o G              % --- 'o' is function composition

IF(b,F,G) : ptrans = LAMBDA q: IF b THEN F(q) ELSE G(q) ENDIF
```

Using the operator PT as implicit type conversion rule, it is possible to mix state transformers and predicate transformers within the same expression. For example, for a state transformer `f` and a predicate transformer `p` we can write `f ++ p` instead of `PT(f) ++ p`.

```
CONVERSION PT                                                        39
```

Assertion and assumption of some Boolean expression `b` are defined using the conditional predicate transformer:

```
assert(b) : ptrans = IF b THEN skip ELSE abort ENDIF                 40
assume(b) : ptrans = IF b THEN skip ELSE magic ENDIF
```

Some typical laws, as used for example in [HJS93, MO96], are as follows:

```
Seq_assoc     : LEMMA   (f ++ g) ++ h = f ++ (g ++ h)                41
Seq_unit_left : LEMMA       skip ++ f = f
Seq_unit_right: LEMMA       f ++ skip = f

Seq_left_zero_bottom: LEMMA (abort ++ f) =  abort
Seq_left_zero_top   : LEMMA (magic ++ f) =  magic

if_true : LEMMA  (IF TRUE THEN f ELSE g ENDIF) = f
if_false: LEMMA (IF FALSE THEN f ELSE g ENDIF) = g
if_same : LEMMA     (IF b THEN f ELSE f ENDIF) = f

assign_cond_rightw: LEMMA
  ((x << e) ++ (IF b THEN f ELSE g ENDIF))
    = IF subst(b, x, e) THEN (x << e) ++ f ELSE (x << e) ++ g ENDIF

assertion_assumption   : LEMMA    assert(b) ++ assume(b) = assert(b)
assumption_assertion   : LEMMA    assume(b) ++ assert(b) = assume(b)
assert_skip            : LEMMA                assert(b) <= skip
assume_skip            : LEMMA                     skip <= assume(b)
combine_assumption     : LEMMA    assume(p) ++ assume(q) = assume(p /\ q)
combine_assertion      : LEMMA    assert(p) ++ assert(q) = assert(p /\ q)
assumption_consequence : LEMMA p <= q IMPLIES assume(q) <= assume(p)
assertion_consequence  : LEMMA p <= q IMPLIES assert(p) <= assert(q)
void_assumption        : LEMMA              assume(TRUE) = skip
void_assertion         : LEMMA              assert(TRUE) = skip

assumption_assign : LEMMA
  (x << e) ++ assume(b) = assume(subst(b, x, e)) ++ (x << e)

assertion_assign : LEMMA
  (x << e) ++ assert(b) = assert(subst(b, x, e)) ++ (x << e)
conditions_cond_top : LEMMA assume(b) ++ F = (IF b THEN F ELSE magic ENDIF)
conditions_cond_bot : LEMMA assert(b) ++ F = (IF b THEN F ELSE abort ENDIF)
```

These kind of laws have heavily been used in a formal verification of compilation theorems of Sampaio's normal form approach [Dol96].

5.4 Structural Operational Semantics

If one slighty changes the presentation of the denotational semantics based on state transformers given in Section 4.2, one obtains a semantics description resembling *structural operational semantics* [Plo81, Kah87].

Rename state transformers *commands*, represented by type `Cmd`, and let a `Configuration` be a pair consisting of a command and the actual program state. Accessor functions for the components of a configuration with respective names are defined.

```
Cmd           : TYPE  = srel                                            42
Configuration : TYPE  = [Cmd, State]

cmd(conf)   : Cmd     = proj_1(conf)
state(conf) : State   = proj_2(conf)
```

The state transition relation `>>` is now defined as a binary relation between (actual) configurations `conf` and (successor) states `s`. A state `s` is an admissible successor of `conf`, written as `conf >> s`, if the evaluation of the actual command `cmd(conf)` in the current state `state(conf)` may end in state `s`:

```
>>(conf,s)  : bool     = member(s,cmd(conf)(state(conf)))               43
```

For presentation purposes we redefine propositional implication:

```
|- : pred[[bool,bool]] = IMPLIES                                        44
|-(p:bool) : bool = p
```

This notation allows a reformulation of the definitions in Section 4.2 as SOS-style rules. For example, the rules for `skip` and sequential composition now read:

```
skip_rule : LEMMA                                                       45
  |- ((skip,s) >> s)

seq_rule : LEMMA
    (f,s) >> t AND (g,t) >> u
  |-
    (f ++ g, s) >> u
```

In order to provide similar rules for assignment, conditional and while-loop, further definitions have to be introduced. In particular, overloading of `>>` is used to denote expression evaluation.

46

```
  BConfiguration : TYPE = [BExpr, State]
  EConfiguration : TYPE = [Expr,  State]

  bconf : VAR BConfiguration
  econf : VAR EConfiguration
  bv    : VAR bool
  v     : VAR value

  bexpr(bconf)  : BExpr   = proj_1(bconf)
  state(bconf)  : State   = proj_2(bconf)
  expr(econf)   : Expr    = proj_1(econf)
  state(econf)  : State   = proj_2(econf)

  >>(bconf,bv) : bool = bv = bexpr(bconf)(state(bconf))
  >>(econf,v)  : bool = v = expr(econf)(state(econf))
```

The following lemmas are again only reformulations of the corresponding denotational semantics definitions. Most of them are simply proved by the strategy (GRIND).

47

```
assign_rule : LEMMA
  declared?(s)(x) IMPLIES
  (
    (e,s) >> v
  |-
    (x << e, s) >> update(s)(x,v)
  )

if_true_rule : LEMMA
    (b,s) >> TRUE AND ((f,s) >> t)
  |-
    (IF b THEN f ELSE g ENDIF, s) >> t

if_false_rule : LEMMA
    (b,s) >> FALSE AND ((g,s) >> t)
  |-
    (IF b THEN f ELSE g ENDIF, s) >> t
```

The proofs of the rules concerning `while` use the corresponding `if`-rules after applying lemma `while_def`, see 22.

48

```
while_false_rule : LEMMA
    (b,s) >> FALSE
  |-
    (while(b,f), s) >> s

while_true_rule : LEMMA
    (b,s) >> TRUE  AND  (f,s) >> t  AND  (while(b,f), t) >> u
  |-
    (while(b,f), s) >> u
```

5.5 Continuations

The style of denotational semantics presented in Section 4.2 is often called *direct semantics* and emphasizes the compositional structure of a language. If control is of concern, for example in imperative languages with unrestricted branching ("goto"), a different style of semantics, namely *continuation semantics* is more appropriate.

The basic elements of continuation semantics are defined at the level of state transformers: a continuation simply *is* a state transformer.

```
continuation : TYPE = srel                                          49
```

The operator `C` describes the continuation-style semantics of some state transformer `f`. It is obtained by simply composing `f` and a given continuation `c`.

```
C(f)(c) : continuation = LAMBDA s: image(c, f(s))                   50

JUDGEMENT C HAS_TYPE [strans -> [strans -> strans]]
```

In the case where the continuation is simply the identity, continuation-style semantics and direct semantics coincide and a transfer lemma is proved providing a way carry over theorems established for one style to the other.

```
cont_singleton : LEMMA                                              51
  C(f)(singleton) = f

transfer: LEMMA
  FORALL (P: pred[srel]):
    P(C(f)(singleton)) = P(f)
```

For completeness, the continuation semantics for simple statements are stated below.

```
skip_continuation : LEMMA                                           52
  C(skip)(c) = c

assign_continuation : LEMMA
  C(x << e)(c)(s) = image(c, ((x << e)(s)))

seq_continuation : LEMMA
  C(f ++ g) = C(f) o C(g)

if_continuation : LEMMA
  C(IF b THEN f ELSE g ENDIF)(c) = IF b THEN C(f)(c) ELSE C(g)(c) ENDIF
```

6. Introducing Variable Declarations

So far, we have only considered statements operating on some abstract state. In this section a concrete implementation of such a program state is described, followed by the definition of the denotational semantics of variable declarations.

The state on which programs operate consists of two parts: a *variable environment* `rho` and a *store* `sigma`. The former is a partial function mapping variable names to abstract locations, the latter maps locations to the values assigned to variables. Since a store should only assign values to those locations actually in use, the state is modeled as a dependent record type:

```
State : TYPE = [# rho : VarEnv, sigma : Store(rho) #]                    53
```

Variable environments are modeled as injective partial functions from the type of variables `Vars` to locations.

```
pre_env : VAR PartialFunction[Vars,Location]                             54

finite_injective_environment?(pre_env) : bool =
    is_finite(image(pre_env,fullset[Vars]):set[Location]) AND
    (FORALL (x,y:Vars, l1,l2:Location):
      x /= y AND pre_env(x)(l1) AND pre_env(y)(l2) IMPLIES l1 /= l2)

VarEnv : TYPE = (finite_injective_environment?)
```

The `Store` maps the locations in use to values.

```
usedlocs(rho) : finite_set[Location] = image(rho,fullset[Vars])         55
UsedLoc(rho)  : TYPE = (usedlocs(rho))

Store(rho)    : TYPE = PartialFunction[UsedLoc(rho),Value]
```

Locations are modelled as natural numbers, and the function `nextloc` takes a finite set of locations and yields a location not already contained in the set.

```
Location : TYPE+ = nat                                                   56

locs : VAR finite_set[Location]

nextloc(locs) : Location =
  IF empty?(locs) THEN 0 ELSE max(locs)+1 ENDIF

nextloc_new : LEMMA
  NOT member(nextloc(locs),locs)
```

The empty state, used as the initial state in the semantics definitions of a programming language, is given by an empty environment and an empty store.

```
empty_env : VarEnv =                                                     57
  LAMBDA (v:Vars): emptyset

empty_store : Store(empty_env) =
  LAMBDA (l:UsedLoc(empty_env)): emptyset

emptystate : [# rho : VarEnv, sigma : Store(rho) #] =
  (# rho   := empty_env, sigma := empty_store #)
```

Based on these definitions, the sets of declared and initialized variables, respectively, can be characterized. A variable `x` is declared in some state `s` if there is an entry for `x` in the current environment `rho(s)`.

```
declared?(s)(x) : bool = singleton?(rho(s)(x))                              58
DeclaredVar(s)  : TYPE = (declared?(s))

newvar?(s)(x) : bool = NOT declared?(s)(x)
NewVar(s)     : TYPE = (newvar?(s))
```

Analogously, declared variables for which there is a value in the store are of type `InitializedVar(s)`.

```
initialized?(s)(x:DeclaredVar(s)) : bool =                                  59
  singleton?(sigma(s)(choose(rho(s)(x))))

InitializedVar(s) : TYPE = (initialized?(s))
```

Next, functions for reading and writing values of variables and for the introduction of new variables are defined. Note that `update` and the lookup function `^` are only defined on declared and initialized variables, respectively.

```
update(s) : [DeclaredVar(s),Value -> State] =                               60
  LAMBDA (x:DeclaredVar(s), v:Value):
    (# rho := rho(s),
       sigma := sigma(s) WITH [(choose(rho(s)(x))) := v] #);

^ : [[s:State, InitializedVar(s)] -> Value] =
  LAMBDA (p:[s:State,InitializedVar(s)]):
    LET (s,x) = p IN
      sigma(s)(choose(rho(s)(x)))

intro(s)(x:NewVar(s)) : State =
  LET newloc = nextloc(usedlocs(rho(s))) IN
    (# rho   := rho(s)   WITH [(x) := newloc],
       sigma := sigma(s) WITH [(newloc) := emptyset] #)
```

The semantics of the declaration of a variable, cf. theory vardecl, is now defined using the `intro` function described above. If the variable `x` is a new variable, `declare(x)(s)` yields a new state with the environment being extended accordingly.

```
declare(x) : srel =                                                         61
  LAMBDA s: IF newvar?(s)(x)
              THEN singleton(intro(s)(x))
              ELSE emptyset
            ENDIF
```

7. Example: A Simple Programming Language

In this section it is described how the definitions presented so far can be applied in order to define the semantics of a simple programming language. We leave variables, values and the various kinds of expressions uninterpreted. They are modeled as theory parameters, as are the semantic functions for expression evaluation.

```
simple_program [Vars, Value, Expr, BExpr : TYPE+,                    62
                (IMPORTING state[Vars,Value])
                eval : [Expr -> [State -> Value]],
                evalB : [BExpr -> [State -> bool]]] : THEORY
```

Variable declarations are simply lists of variables, and the abstract syntax of commands is defined by an inductive datatype. Programs are then pairs consisting of a declaration and a command.

```
VarDecl : TYPE = list[Vars]                                          63

SimpleCommand : DATATYPE
BEGIN
  skip : skip?
  seq(first,second:SimpleCommand) :seq?
  assign(varid:Vars, exp:Expr) : assign?
  if_(ifcond: BExpr, thn, els: SimpleCommand) : if?
  while(whilecond: BExpr, body: SimpleCommand) : while?
END SimpleCommand

SimpleProgram : TYPE = [# decl : VarDecl, body : SimpleCommand #]
```

By importing the definitions presented in the previous sections, the semantics of the commands are defined inductively using the corresponding semantic functions.

```
IMPORTING semantics[Vars,Value]                                      64

c : VAR SimpleCommand

[||] : [Expr -> [State -> Value]] = eval
[||] : [BExpr -> [State -> bool]] = evalB

[||](c) : RECURSIVE srel =       % --- prefix variant of [| c |]
  CASES c OF
    skip       : skip,
    seq(f,g)   : [| f |] ++ [| g |],
    assign(x,e) : x << [| e |],
    if_(b,f,g)  : IF [| b |] THEN [| f |] ELSE [| g |] ENDIF,
    while(b,f)  : while([| b |],[| f |])
  ENDCASES
    MEASURE c BY <<
```

The semantic function for declarations creates a new entry for each of the variables in sequence.

```
[||](d:VarDecl) : RECURSIVE srel =                                              65
  IF null?(d) THEN skip ELSE declare(car(d)) ++ [| cdr(d) |] ENDIF
    MEASURE length(d)
```

The semantics of a program is then defined straightforward: evaluating the declarations in the empty state leads to an environment in which the commands are evaluated.

```
p : VAR SimpleProgram                                                           66

[||](p) : set[State] =
  [| body(p) |]([| decl(p) |](emptystate))
```

8. Conclusions

We have presented a uniform formalization in PVS of various kinds of semantics of the basic constructs in imperative programming languages. Based on a comprehensive development of fixed point theory, the denotational semantics of the constructs were defined as state transformers. These state transformers induce corresponding predicate transformers, providing a means to formally derive both weakest liberal precondition semantics and axiomatic semantics in the style of Hoare. Moreover, algebraic laws as used in refinement calculus proofs have been validated at the level of predicate transformers. A simple reformulation of the state transformers semantics yields a continuation-style semantics and rules similar to those used in structural operational semantics.

Since the various semantics are derived rather than defined, many interchangeable forms of semantics are available. Thus, formal reasoning about both programs and their compilation can be carried out within the same framework.

The theories comprise semantic definitions for the basic constructs of imperative programming languages. This will be further extended by additional definitions for other constructs such as (mutually) recursive procedures in order to be applicable to a wider range of programming languages. Besides the fact that this requires some additional technical work concerning the modelling of underlying mathematical structures, such as (simultaneous) fixpoints of a set of functions, this should not impose major difficulties.

The formalization presented in this paper provides the foundations on which formal specification of programming language semantics and mechanized verification of compilation steps are carried out within the *Verifix* project. The theories have been used in the specification and verification of a generic compilation of imperative programming constructs [DvHPR96].

Future investigations will be concerned with how moving between different styles of semantics can be exploited in verifying compilation steps. For example, very often the semantics of the source and target language of a compiler are given in a denotational and an operational style, respectively. One way of proving the correctness of the compiler in such a case is to derive first a denotational semantics of the target language from its operational semantics [NN92]. We conjecture that a systematic support of these kinds of derivations would help to reduce the proof effort of such correctness proofs.

References

[Bac80] R. J. R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Centre Tracts*. Mathematical Centre, Amsterdam, 1980.

[Bac81] R. J. R. Back. On Correct Program Refinements. *Journal of Computer and System Sciences*, 23(1):49–68, August 1981.

[Bac89] R. J. R. Back. Refinement Calculus Part I: Sequential Nondeterministic Programs. In J. W. de Bakker and W.-P. de Roever, editors, *Stepwise Refinement of Distributed Systems. REX-Workshop*, volume 430 of *LNCS*. Springer Verlag, 1989.

[BDvH+96] 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.

[Bes95] E. Best. *Semantik – Theorie sequentieller und paralleler Programmierung.* Lehrbuch Informatik. Vieweg-Verlag, 1995.

[BvW90] R. J. R. Back and J. v. Wright. Refinement Concepts Formalised in Higher Order Logic. *Formal Aspects of Computing*, 2:247–272, 1990.

[Cam90] A. J. Camilleri. Mechanizing CSP Trace Theory in Higher Order Logic. *IEEE Transactions of Software Engineering*, 16(9):993–1004, 1990.

[Dij71] E. W. Dijkstra. Notes on Structured Programming. In E. D. Dahl and C. Hoare, editors, *Structured Programming.* Academic Press, 1971.

[Dol96] Axel Dold. A Formalization of the Normal Form Approach to Compilation. *Verifix* working paper [Verifix / Uni Ulm / 9.1], Universität Ulm, July 1996.

[DvHPR96] A. Dold, F.W. von Henke, H. Pfeifer, and H. Rueß. Generic Specification of Correct Compilation. Ulmer Informatik-Berichte 96-12, Universität Ulm, December 1996.

[Flo67] R. W. Floyd. Assigning Meaning to Programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, pages 19–32. American Math Society, Providence, Rhode Island, 1967.

[Gor89] M. J. C. Gordon. Mechanizing Programming Logics in Higher Order Logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Theorem Proving.* Springer-Verlag, 1989.

[HHJ+87] C. A. R. Hoare, I. J. Hayes, H. Jifeng, C. C. Morgan, A . W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Communications of the ACM*, 8(8):672–686, August 1987.

[HJS93] C. A. R. Hoare, H. Jifeng, and A. Sampaio. Normal Form Approach to Compiler Design. *Acta Informatica*, 30:701–739, 1993.

[HM95] P. V. Homeier and D. F. Martin. A Mechanically Verified Verification Condition Generator. *The Computer Journal*, 38(1), 1995.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *CACM*, 12:576–580, 1969.

[Kah87] G. Kahn. Natural Semantics. In *Proc. STACS '87*, volume 247 of *LNCS*, pages 22–39, berlin, 1987. Springer.

[Mas87] I. A. Mason. Hoare's Logic in the LF. Technical Report 87-32, Laboratory for Foundations of Computer Science, University of Edinburgh, 1987.

[MO96] M. Müller-Olm. *Modular Compiler Verification.* PhD thesis, Christian Albrechts Universität zu Kiel, 1996.

[Nip96] Tobias Nipkow. Winskel is (almost) Right: Towards a Mechanized Semantics Textbook. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *Lecture Notes in Computer Science*, pages 180–192. Springer Verlag, 1996.

[NN92] H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction.* Wiley Professional Computing. Wiley, 1992.

[ORSvH95] 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.

[Plo81] G. Plotkin. A Structural Approach to Operational Semantics. Report DAIMI-FN-19, Computer Science Department, Aarhus University, 1981.

[Sch88] D. A. Schmidt. *Denotational Semantics.* Wm. C. Brown Publishers, Dubuque, Iowa, 1988.

[Sok87] S. Sokolowski. Soundness of Hoare's Logic: An Automated Proof Using LCF. *ACM Transactions on Programming Languages and Systems*, 9(1):100–120, 1987.

[SS72] D. Scott and D. Strachey. Towards a Mathematical Semantics for Computer Languages. In J. Fox, editor, *Computers and Automata*, pages 19–46. J. Wiley, New York, 1972.

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, 1977.

[Ten76] R. D. Tennent. The Denotational Semantics of Programming Languages. *CACM*, 19:437–453, 1976.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages.* Foundations of Computing Series. MIT Press, Cambridge, Massachusetts, 1993.

[Wir71] N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14:221–227, 1971.

Appendix: PVS Source Files

A. Denotational Semantics

A.1 State transformers

```
srel [sigma, tau: TYPE+] : THEORY

BEGIN

  FP : LIBRARY = "./Fixpoints/"

  % -------- (Nondeterministic) State transformers

  IMPORTING relation[sigma,tau], FP@predicate_cpo[tau]

  srel : TYPE = Relation[sigma,tau]  % i.e. srel = [sigma -> set[tau]]

  s   : VAR sigma
  S   : VAR set[sigma]
  T   : VAR set[tau]
  f,g : VAR srel

  % -------- least state transformers

  abort : srel = LAMBDA s: emptyset;

  % -------- Ordering on state transformers:
  %          the next IMPORT defines a partial order <= on state transformers
  %          as f <= g  :<=>  FORALL s: f(s) IMPLIES g(s)

  IMPORTING FP@dcpo_to_precpo[sigma,set[tau],pointwise[tau,bool,=>].<=]
  IMPORTING FP@cpo_defs[srel]

  JUDGEMENT <= HAS_TYPE pCPO[srel]
  JUDGEMENT abort HAS_TYPE (bottom?(<=):pred[srel])

  % -------- importing definitions and theorems about fixpoints on srels

  IMPORTING FP@fixpoints_cont[srel,<=,abort]

  % -------- Deterministic state transformers

  strans : TYPE = PartialFunction[sigma,tau]

END srel
```

A.2 Program State

Variable environment

```
var_environment [Vars : TYPE+] : THEORY

BEGIN

  IMPORTING location, relation, finite_sets@finite_sets_def[Location]
```

```
  locs    : VAR set[Location]
  pre_env : VAR PartialFunction[Vars,Location]

  finite_injective_environment?(pre_env) : bool =
    is_finite(image(pre_env,fullset[Vars]):set[Location]) AND
    (FORALL (x,y:Vars, l1,l2:Location):
      x /= y AND pre_env(x)(l1) AND pre_env(y)(l2) IMPLIES l1 /= l2)


  VarEnv : TYPE = (finite_injective_environment?)

END var_environment
```

Locations

```
location : THEORY

BEGIN

  Location : TYPE+ = nat

  IMPORTING finsets[Location,<=]

  locs : VAR finite_set[Location]

  nextloc(locs) : Location =
    IF empty?(locs) THEN 0 ELSE maxrec(locs)+1 ENDIF


  % -------- nextloc yields a new location

  nextloc_max : LEMMA
    FORALL (l:(locs)): l < nextloc(locs)

  nextloc_new : LEMMA
    NOT locs(nextloc(locs))

END location
```

Program State

```
state [Vars, Value : TYPE+] : THEORY

BEGIN

  IMPORTING var_environment[Vars], deterministic[Value]
  CONVERSION singleton, select[Location], select[Value]

  rho : VAR VarEnv
  x   : VAR Vars
  v   : VAR Value
  l   : VAR Location


  usedlocs(rho) : finite_set[Location] = image(rho,fullset[Vars])
```

```
UsedLoc(rho) : TYPE = (usedlocs(rho))

Store(rho) : TYPE = PartialFunction[UsedLoc(rho),Value]

empty_env : VarEnv =
  LAMBDA (v:Vars): emptyset[Location]

empty_store : Store(empty_env) =
  LAMBDA (l:UsedLoc(empty_env)): emptyset

emptystate : [# rho : VarEnv, sigma : Store(rho) #] =
  (# rho   := empty_env, sigma := empty_store #)

State : TYPE = [# rho : VarEnv, sigma : Store(rho) #]

JUDGEMENT emptystate HAS_TYPE State

s  : VAR State

% -------- declared variable: has a location assigned to it

declared?(s)(x) : bool = singleton?(rho(s)(x))

DeclaredVar(s) : TYPE = (declared?(s))

newvar?(s)(x) : bool = NOT declared?(s)(x)

NewVar(s) : TYPE = (newvar?(s))

% -------- (declared) variable is initialized, if there's a value
%          assigned to it.

initialized?(s)(x:DeclaredVar(s)) : bool =
  singleton?(sigma(s)(select(rho(s)(x))))

InitializedVar(s) : TYPE = (initialized?(s))

% -------- update value of declared variables

update(s) : [DeclaredVar(s),Value -> State] =
  LAMBDA (x:DeclaredVar(s), v:Value):
    (# rho := rho(s),
       sigma := sigma(s) WITH [(select(rho(s)(x))) := v] #);

update_initializes : LEMMA
  FORALL (s: State, x:DeclaredVar(s), v:Value):
    initialized?(update(s)(x,v))(x);

% -------- read value of declared variables

^ : [[s:State, InitializedVar(s)] -> Value] =
  LAMBDA (p:[s:State,InitializedVar(s)]):
    LET s = proj_1(p), x = proj_2(p) IN
      sigma(s)(select(rho(s)(x)))
% -------- properties
```

```
  update_access: LEMMA
    FORALL (s: State, x:DeclaredVar(s), v:Value):
      update(s)(x,v)^x = v

  equal_updates: LEMMA
    FORALL (s: State, x:DeclaredVar(s), v1,v2:Value):
      update(update(s)(x,v2))(x,v1)^x = v1

  nonequal_updates: LEMMA
    FORALL (s: State, x1:DeclaredVar(s), x2:DeclaredVar(s), v1,v2:Value):
      x1 /= x2 IMPLIES update(update(s)(x2, v2))(x1, v1)^x2 = v2

  % -------- introduce new variable

  extend_varenv : LEMMA
    FORALL (s:State,x:NewVar(s)):
      LET newloc = nextloc(usedlocs(rho(s))) IN
        finite_injective_environment?(rho(s) WITH [(x) := newloc])

  intro(s)(x:NewVar(s)) : State =
    LET newloc = nextloc(usedlocs(rho(s))) IN
      (# rho   := rho(s)   WITH [(x) := newloc],
         sigma := sigma(s) WITH [(newloc) := emptyset] #)


  % -------- Substitution

  subst(p: pred[State], x:Vars, sv: [State -> Value]) : pred[State] =
    LAMBDA s: IF declared?(s)(x) THEN p(update(s)(x,sv(s))) ELSE TRUE ENDIF

  % -------- Independence

  independent?(p: pred[State], x: Vars): bool =
    FORALL (s: State, sv:[State -> Value]): subst(p,x,sv)(s) = p(s)

END state
```

A.3 State Transformer Semantics

Boolean Expressions

```
bexpr[sigma: TYPE+]: THEORY

BEGIN

  FP : LIBRARY = "./Fixpoints/"

  IMPORTING FP@predicates[sigma]

  BExpr : TYPE = pred[sigma]

END bexpr
```

Expressions

```
expr[state, value: TYPE+]: THEORY

BEGIN

  Expr : TYPE = [state -> value]

  IMPORTING bexpr[state]

END expr
```

Skip

```
skip [sigma : TYPE+] : THEORY

BEGIN

  IMPORTING srel[sigma,sigma]

  s : VAR sigma

  % -------- Do Nothing

  skip : srel = LAMBDA s: singleton(s)

  JUDGEMENT skip HAS_TYPE strans

END skip
```

Assignment

```
assignment [Vars, Value, sigma: TYPE+,
            declared? : [sigma -> pred[Vars]],
            update : [s:sigma -> [(declared?(s)), Value -> sigma]]
           ] : THEORY
BEGIN

  IMPORTING expr[sigma,Value], srel[sigma,sigma]

  s       : VAR sigma
  x       : VAR Vars
  e       : VAR Expr

  % -------- Assignments

  <<(x,e) : srel =
    LAMBDA s: IF declared?(s)(x)
                THEN singleton(update(s)(x,e(s)))
                ELSE emptyset
              ENDIF

  JUDGEMENT << HAS_TYPE [Vars,Expr -> strans]

END assignment
```

Sequential Composition

```
sequence [sigma : TYPE+] : THEORY

BEGIN

  IMPORTING srel[sigma,sigma]

  s            : VAR sigma
  f,g,h        : VAR srel
  f1,f2,g1,g2 : VAR srel

  % -------- Sequencing

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

  sequence_strans_closed : LEMMA
    FORALL (f,g:strans): deterministic?((f ++ g)(s))

  JUDGEMENT ++ HAS_TYPE [strans, strans -> strans]

  sequence_monotonic_left : LEMMA
    f <= g IMPLIES f ++ h <= g ++ h

  sequence_monotonic_right : LEMMA
    f <= g IMPLIES h ++ f <= h ++ g

  sequence_monotonic : LEMMA
    f1 <= f2 AND g1 <= g2 IMPLIES f1 ++ g1 <= f2 ++ g2

END sequence
```

Conditional and While Loop

```
control_structures [sigma : TYPE+] : THEORY

BEGIN

  IMPORTING sequence[sigma], skip[sigma], bexpr[sigma]

  b     : VAR BExpr
  f,g,h : VAR srel
  s     : VAR sigma

  % -------- Conditionals

  IF(b, f, g) : srel =
    LAMBDA s: IF b(s) THEN f(s) ELSE g(s) ENDIF;

  IF_monotonic_left : LEMMA
    f <= g IMPLIES
      IF b THEN f ELSE h ENDIF  <=  IF b THEN g ELSE h ENDIF

  IF_monotonic_right : LEMMA
    f <= g IMPLIES
     IF b THEN h ELSE f ENDIF  <=  IF b THEN h ELSE g ENDIF
```

```
  IF_strans_closed : LEMMA
    FORALL (f,g:strans): deterministic?((IF b THEN f ELSE g ENDIF)(s))


  % -------- while loop

  while_functional_monotonic : LEMMA
    monotonic?(LAMBDA (x:srel): IF b THEN f ++ x ELSE skip ENDIF)

  while(b,f) : srel =
    mu! (x:srel): IF b THEN f ++ x ELSE skip ENDIF

  % -------- admissibility needed in proof of judgement

  while_strans_adm : LEMMA
    admissible?(LAMBDA (y: srel): FORALL (s: sigma): deterministic?(y(s)))

  while_strans_closed : LEMMA
    FORALL (f:strans): FORALL (s:sigma): deterministic?(while(b,f)(s))

  JUDGEMENT while HAS_TYPE [[BExpr, strans] -> strans]


  while_def : LEMMA
    while(b,f) = (IF b THEN f ++ while(b,f) ELSE skip ENDIF)

  while_montonic : LEMMA
    f <= g IMPLIES while(b,f) <= while(b,g)

END control_structures
```

Semantics of Simple Statements

```
statements [Vars, Value : TYPE+] : THEORY

BEGIN

  IMPORTING state[Vars,Value]
  IMPORTING control_structures[State]
  IMPORTING sequence[State]
  IMPORTING assignment[Vars, Value, State, declared?, update]

END statements
```

A.4 Predicates

```
predicates [sigma: TYPE+]: THEORY

BEGIN

  % -------- Lifting boolean connectives to predicates over sigma

  s     : VAR sigma
  p,q,b : VAR pred[sigma]
  S     : VAR set[sigma]
```

```
  TRUE     :pred[sigma] = LAMBDA s: TRUE;
  FALSE    :pred[sigma] = LAMBDA s: FALSE;
  NOT(p)   :pred[sigma] = LAMBDA s: NOT(p(s));
  /\(p, q) :pred[sigma] = LAMBDA s: p(s) AND q(s);
  \/(p, q) :pred[sigma] = LAMBDA s: p(s) OR q(s);
  =>(p, q) :pred[sigma] = LAMBDA s: p(s) IMPLIES q(s);
  <=>(p, q):pred[sigma] = LAMBDA s: p(s) IFF q(s);

  /\(PP: set[pred[sigma]]): pred[sigma] = LAMBDA s: FORALL (p: (PP)): p(s);
  \/(PP: set[pred[sigma]]): pred[sigma] = LAMBDA s: EXISTS (p: (PP)): p(s);

  IF(b,p,q) : pred[sigma] = (LAMBDA s: IF b(s) THEN p(s) ELSE q(s) ENDIF);

END predicates
```

Predicate CPOs

```
predicate_cpo  [ sigma: TYPE+ ]: THEORY

BEGIN

  % -------- booleans with implication form a cpo:

  IMPORTING cpo_defs
  IMPORTING predicates[sigma]
  IMPORTING bool_cpo

  bottom: pred[sigma] = FALSE;
  top   : pred[sigma] = TRUE


  % -------- Ordering on predicates:
  %          the next IMPORT defines a partial order <= on predicates as
  %          p <= q  :<=>  FORALL s: p(s) IMPLIES q(s)

  % -------- predicates are functions from a type (i.e. a discrete cpo) sigma
  %          into a cpo, viz. bool, hence predicates with <= form a cpo.

  IMPORTING dcpo_to_precpo[sigma,bool,=>]

  bottom_pred: LEMMA
    bottom?[[ sigma -> bool]](pointwise[sigma,bool,=>].<=)(bottom)

  IMPORTING cpo[ pred[sigma],<=,bottom]

  PP: VAR set[ pred[sigma]]

  pred_lub: LEMMA lub?(\/( PP),PP)

  pred_lub_exists: LEMMA lub_exists?(PP)

  pred_lub_is: LEMMA lub(PP) = \/(PP)

END predicate_cpo
```

A.5 Predicate Transformers

```
ptrans [sigma, tau: TYPE+]:THEORY

BEGIN

  FP : LIBRARY = "./Fixpoints/"

  predS : THEORY = FP@predicate_cpo[sigma]
  predT : THEORY = FP@predicate_cpo[tau]

  % -------- Ordering
  %          next IMPORT defines partial order <= on predicates transformers
  %          as F <= G  :<=>  FORALL q: F(q) IMPLIES G(q)

  IMPORTING FP@function_cpo[pred[tau],pred[sigma],pointwise.<=,bottom]
  IMPORTING FP@monotonic[pred[tau],<=,pred[sigma],<=]

  % ---------- predicate transformers and monotonic predicate transformers

  ptrans : TYPE = [pred[tau] -> pred[sigma]]
  MPT    : TYPE = {F:ptrans | monotonic?(F)}

  JUDGEMENT <= HAS_TYPE pCPO[ptrans]

  % -------- Basic Predicate Transformers

  q   : VAR pred[tau];  y: VAR sigma;  M: VAR set[ptrans]

  abort   : ptrans = LAMBDA q: FALSE;
  magic   : ptrans = LAMBDA q: TRUE;
  NOT(F)  : ptrans = LAMBDA q: NOT(F(q));
  /\(F, G): ptrans = LAMBDA q: F(q) /\ G(q); % demonic choice
  \/(F, G): ptrans = LAMBDA q: F(q) \/ G(q); % angelic choice
  =>(F, G): ptrans = LAMBDA q: F(q) => G(q);

  IMPORTING FP@cpo_defs[ptrans]
  JUDGEMENT abort HAS_TYPE (bottom?(<=):pred[ptrans])

  % -------- Generalized Meet and Join

  /\(M): ptrans = LAMBDA q: LAMBDA y: FORALL F: M(F) IMPLIES F(q)(y);
  \/(M): ptrans = LAMBDA q: LAMBDA y: EXISTS F: M(F) AND F(q)(y);

END ptrans
```

A.6 Predicate Transformer Semantics

Monotonic Predicate Transformers

```
mpt [sigma, tau: TYPE+] : THEORY

BEGIN

  IMPORTING ptrans[sigma,tau], bexpr[sigma]
```

```
  % -------- define monotonic predicate transformer induced by srel:

  IMPORTING srel[sigma,tau]

  b   : VAR BExpr;   T   : VAR set[tau]
  f,g : VAR srel;    F,G : VAR ptrans

  PT(f: srel) : ptrans = LAMBDA T: inverse_image(f,T)

  JUDGEMENT PT HAS_TYPE [srel -> MPT]
  CONVERSION PT

  % -------- some properties of PT

  lifting : LEMMA
    f <= g IFF PT(g) <= PT(f)

  PT_universally_conjunctive : LEMMA
    FORALL (M:set[pred[tau]]):
      PT(f)(/\(M)) = /\({p:pred[sigma] | EXISTS (m:(M)): p = PT(f)(m)})

  % -------- monotonic pred. trans. as used e.g. by Back, Sampaio and others

  JUDGEMENT abort HAS_TYPE MPT
  JUDGEMENT magic HAS_TYPE MPT
  JUDGEMENT /\    HAS_TYPE [MPT,MPT -> MPT]
  JUDGEMENT \/    HAS_TYPE [MPT,MPT -> MPT]

END mpt
```

Monotonic Predicate Transformers for Statements

```
stmt_mpt [sigma : TYPE+] : THEORY

BEGIN

  IMPORTING mpt[sigma,sigma], skip[sigma]

  b   : VAR BExpr
  F,G : VAR ptrans
  q   : VAR pred[sigma]

  % -------- do nothing

  skip : ptrans = PT(skip);

  JUDGEMENT skip HAS_TYPE MPT

  %-------- sequential composition

  ++(F,G) : ptrans = F o G

  JUDGEMENT ++ HAS_TYPE [MPT,MPT -> MPT]
```

```
  % -------- conditional

  IF(b,F,G) : ptrans = LAMBDA q: IF b THEN F(q) ELSE G(q) ENDIF

END stmt_mpt
```

B. Various Styles of Semantics

B.1 Weakest Precondition

```
wp [Vars, Value: TYPE+] : THEORY

BEGIN

  IMPORTING statements[Vars, Value], mpt

  f,g,h : VAR srel
  x     : VAR Vars
  e     : VAR Expr
  b     : VAR BExpr
  p,q,r : VAR pred[State]


  % -------- Weakest Precondition

  wp(f,q) : pred[State] = PT(f)(q);


  % -------- properties

  wp_char : LEMMA
    wp(f, q) = \/({p | p <= PT(f)(q)})

  wp_skip : LEMMA
    wp(skip, q) = q

  wp_assign : LEMMA
    wp(x << e, q) = subst(q, x, e)

  wp_if : LEMMA
    wp(IF b THEN f ELSE g ENDIF, q)
      = (IF b THEN wp(f, q) ELSE wp(g, q) ENDIF)

  wp_seq : LEMMA
    wp(f ++ g, q) = wp(f,wp(g,q))

  wp_while : LEMMA
    wp(while(b,f), q)
      = (IF b THEN wp(f ++ while(b,f),q) ELSE q ENDIF)

END wp
```

B.2 Hoare Logic

```
hoare_logic[Vars, Value: TYPE+] : THEORY

BEGIN

  IMPORTING statements[Vars, Value]

  b          : VAR BExpr
  e          : VAR Expr
  x          : VAR Vars
  p, p1, p2,
  q, q1, q2,
  r          : VAR pred[State]
  f, g, h    : VAR srel
  s          : VAR State


  % -------- Hoare triple

  |=(p, f, q) : bool = image(f,p) <= q


  % -------- Characteristic Lemmas

  precondition_strengthening : LEMMA
      p1 <= p AND |=(p, f, q)
    IMPLIES
      |=(p1, f, q)

  postcondition_weakening : LEMMA
      |=(p, f, q) AND q <= q1
    IMPLIES
      |=(p, f, q1)

  rule_of_consequence : COROLLARY   % --- combination of the above
      p <= p1 AND |=(p1, f, q1) AND q1 <= q
    IMPLIES
      |=(p, f, q)

  % -------- rules for statements

  skip_rule : LEMMA
    |=(p, skip, p)

  assign_rule : LEMMA
    |=(subst(p, x, e), (x << e), p)

  seq_rule : LEMMA
      |=(p, f, q) AND |=(q, g, r)
    IMPLIES
      |=(p, f ++ g, r)

  if_rule : LEMMA
      |=(p /\ b, f, q) AND |=(p /\ NOT(b), g, q)
    IMPLIES
```

```
      |=(p, (IF b THEN f ELSE g ENDIF), q)

  % -------- needed in proof of 'while_rule'

  while_adm : LEMMA
    admissible?(LAMBDA f: |=(p, f, p /\ NOT(b)))

  while_rule : LEMMA
      |=(p /\ b, h, p)
    IMPLIES
      |=(p, while(b, h), p /\ NOT(b))

END hoare_logic
```

Equivalence of Hoare Logic and Weakest Precondition Semantics

```
wp_hoare [Vars, Value: TYPE+] : THEORY

BEGIN

  IMPORTING wp[Vars,Value]
  IMPORTING hoare_logic[Vars,Value]

  f   : VAR srel
  p,q : VAR pred[State]


  % -------- Equivalence of Hoare Calculus and Weakest Precondition

  hoare_logic_equiv_wp : THEOREM
     |=(p, f, q) = (p <= wp(f, q))

END wp_hoare
```

B.3 Algebraic Laws

```
laws[vars, value: TYPE+]: THEORY

BEGIN

  IMPORTING statements[vars, value]
  IMPORTING assert_assume[State]

  b         : VAR BExpr
  x         : VAR vars
  s         : VAR State
  e, e1, e2: VAR Expr
  f, g, h   : VAR srel
  p, q, r   : VAR pred[State]
  F,G       : VAR ptrans

  % -------- (Sequential Composition):

  Seq_assoc     : LEMMA   (f ++ g) ++ h = f ++ (g ++ h)
  Seq_unit_left : LEMMA       skip ++ f = f
```

```
  Seq_unit_right: LEMMA       f ++ skip = f

  % -------- (Chaos, Miracle and Top)

  Seq_left_zero_bottom: LEMMA (abort ++ f) =  abort
  Seq_left_zero_top   : LEMMA (magic ++ f) =  magic

  % -------- (Conditionals)

  if_true : LEMMA  (IF TRUE THEN f ELSE g ENDIF) = f
  if_false: LEMMA (IF FALSE THEN f ELSE g ENDIF) = g
  if_same : LEMMA     (IF b THEN f ELSE f ENDIF) = f

  seq_cond_leftw: LEMMA
    (IF b THEN f ELSE g ENDIF) ++ h = (IF b THEN f ++ h ELSE g ++ h ENDIF)

  assign_cond_rightw: LEMMA          % nur deterministische Zuweisung !!
    ((x << e) ++ (IF b THEN f ELSE g ENDIF))
      = IF subst(b, x, e) THEN (x << e) ++ f ELSE (x << e) ++ g ENDIF

  % -------- (Loops and Fixpoints)

  while_def: LEMMA
    while(b,f) = (IF b THEN f ++ while(b,f) ELSE skip ENDIF)

  % -------- (Assumptions and Assertions)

  assertion_assumption   : LEMMA    assert(b) ++ assume(b) = assert(b)
  assumption_assertion   : LEMMA    assume(b) ++ assert(b) = assume(b)
  assert_skip            : LEMMA                assert(b) <= skip
  assume_skip            : LEMMA                     skip <= assume(b)
  combine_assumption     : LEMMA    assume(p) ++ assume(q) = assume(p /\ q)
  combine_assertion      : LEMMA    assert(p) ++ assert(q) = assert(p /\ q)
  assumption_consequence : LEMMA p <= q IMPLIES assume(q) <= assume(p)
  assertion_consequence  : LEMMA p <= q IMPLIES assert(p) <= assert(q)
  void_assumption        : LEMMA              assume(TRUE) = skip
  void_assertion         : LEMMA              assert(TRUE) = skip

  assumption_assign : LEMMA
    (x << e) ++ assume(b) = assume(subst(b, x, e)) ++ (x << e)

  assertion_assign : LEMMA
    (x << e) ++ assert(b) = assert(subst(b, x, e)) ++ (x << e)

  cond_assertion_then : LEMMA
    PT(IF b THEN f ELSE g ENDIF) = (IF b THEN assert(b) ++ f ELSE g ENDIF)

  cond_assertion_else : LEMMA
    PT(IF b THEN f ELSE g ENDIF) =
      IF b THEN f ELSE assert(NOT(b)) ++ g ENDIF

  conditions_cond_top : LEMMA assume(b) ++ F = (IF b THEN F ELSE magic ENDIF)
  conditions_cond_bot : LEMMA assert(b) ++ F = (IF b THEN F ELSE abort ENDIF)

END laws
```

B.4 SOS

```
sos [vars,value : TYPE+] : THEORY

BEGIN

  IMPORTING statements[vars,value]

  Cmd        : TYPE = srel

  Configuration  : TYPE = [Cmd, State]
  BConfiguration : TYPE = [BExpr,State]
  EConfiguration : TYPE = [Expr,State]

  b     : VAR BExpr;   e : VAR Expr;    x : VAR vars
  bv    : VAR bool;    v : VAR value
  s,t,u : VAR State;   c,f,g : VAR Cmd
  conf  : VAR Configuration
  bconf : VAR BConfiguration
  econf : VAR EConfiguration

  cmd(conf) : Cmd = proj_1(conf)
  state(conf)   : State   = proj_2(conf)

  bexpr(bconf)  : BExpr   = proj_1(bconf)
  state(bconf)  : State   = proj_2(bconf)
  expr(econf)   : Expr    = proj_1(econf)
  state(econf)  : State   = proj_2(econf);

  >>(bconf,bv) : bool = bv = bexpr(bconf)(state(bconf));
  >>(econf,v)  : bool = v = expr(econf)(state(econf));
  >>(conf,s)   : bool = member(s,cmd(conf)(state(conf)));

  |- : pred[[bool,bool]] = IMPLIES;
  |-(p:bool) : bool = p;

  skip_rule : LEMMA
    |- ((skip,s) >> s)

  assign_rule : LEMMA
    declared?(s)(x) IMPLIES
    (
      (e,s) >> v
    |-
      (x << e, s) >> update(s)(x,v)
    )

  seq_rule : LEMMA
      (f,s) >> t AND (g,t) >> u
    |-
      (f ++ g, s) >> u

  if_true_rule : LEMMA
      (b,s) >> TRUE AND ((f,s) >> t)
    |-
      (IF b THEN f ELSE g ENDIF, s) >> t
```

```
  if_false_rule : LEMMA
      (b,s) >> FALSE AND ((g,s) >> t)
    |-
      (IF b THEN f ELSE g ENDIF, s) >> t

  while_false_rule : LEMMA
      (b,s) >> FALSE
    |-
      (while(b, f), s) >> s

  while_true_rule : LEMMA
      (b,s) >> TRUE  AND  (f,s) >> t  AND  (while(b,f), t) >> u
    |-
      (while(b,f), s) >> u

END sos
```

B.5 Continuation

```
continuation[sigma: TYPE+]: THEORY

BEGIN

  IMPORTING srel[sigma, sigma]

  continuation : TYPE = srel

  s : VAR sigma
  c : VAR continuation
  f : VAR srel

  C(f)(c) : continuation = LAMBDA s: image(c, f(s))

  cont_singleton : LEMMA
    C(f)(singleton) = f

  transfer: LEMMA
    FORALL (P: pred[srel]):
      P(C(f)(singleton)) = P(f)

END continuation

continuation_lems[vars, value: TYPE+]: THEORY

BEGIN

  IMPORTING statements[vars, value]
  IMPORTING continuation[State]

  x   : VAR vars; b : VAR BExpr; e : VAR Expr
  s   : VAR State
  c   : VAR continuation
  p,q : VAR pred[State]
  f,g : VAR srel

  skip_continuation_eq: LEMMA
```

```
    C(skip)(c) = c

  assign_continuation_eq: LEMMA
    C(x << e)(c)(s) = image(c, ((x << e)(s)))

  seq_continuation_eq: LEMMA
    C(f ++ g) = C(f) o C(g)

  if_continuation_eq: LEMMA
    C(IF b THEN f ELSE g ENDIF)(c) = (IF b THEN C(f)(c) ELSE C(g)(c) ENDIF);

END continuation_lems
```

B.6 Putting it Together

```
semantics [Vars, Value: TYPE+] : THEORY

BEGIN

  IMPORTING sos[Vars,Value],
            continuation_lems[Vars,Value],
            wp_hoare[Vars,Value],
            laws[Vars,Value]

END semantics
```

C. A Simple Programming Language

C.1 Simple Commands

```
simple_commands [Vars, Value, Expr, BExpr: TYPE+,
                 (IMPORTING state[Vars,Value])
                 eval : [Expr -> [State -> Value]],
                 evalB : [BExpr -> [State -> bool]]
                ] : THEORY
BEGIN

  % ---------- syntax

  SimpleCommand : DATATYPE
  BEGIN
    skip : skip?
    seq(first,second:SimpleCommand) :seq?
    assign(varid:Vars, exp:Expr) : assign?
    if_(ifcond: BExpr, thn, els: SimpleCommand) : if?
    while(whilecond: BExpr, body: SimpleCommand) : while?
  END SimpleCommand

  % ---------- semantics

  IMPORTING semantics[Vars,Value]

  [||] : [Expr -> [State -> Value]] = eval
```

```
  [||] : [BExpr -> [State -> bool]] = evalB

  [||](c:SimpleCommand) : RECURSIVE srel =
    CASES c OF
      skip        : skip,
      seq(f,g)    : [| f |] ++ [| g |],
      assign(x,e) : x << [| e |],
      if_(b,f,g)  : IF [| b |] THEN [| f |] ELSE [| g |] ENDIF,
      while(b,f)  : while([| b |],[| f |])
    ENDCASES MEASURE c BY <<

END simple_commands
```

C.2 Simple Declarations

```
simple_declarations [Vars, Value : TYPE+] : THEORY

BEGIN

  % ---------- syntax

  VarDecl : TYPE = list[Vars]

  % ---------- semantics

  IMPORTING state[Vars,Value]
  IMPORTING vardecl[Vars, Value, State, newvar?, intro]
  CONVERSION deterministic[State].select

  d : VAR VarDecl
  s : VAR State

  [||](d:VarDecl) : RECURSIVE srel =
    IF null?(d) THEN skip ELSE declare(car(d)) ++ [| cdr(d) |] ENDIF
      MEASURE length(d)

END simple_declarations
```

C.3 The Language

```
simple_program [Vars, Value, Expr, BExpr: TYPE+,
                (IMPORTING state[Vars,Value])
                eval : [Expr -> [State -> Value]],
                evalB : [BExpr -> [State -> bool]]] : THEORY
BEGIN

  % ---------- syntax

  IMPORTING simple_commands[Vars,Value,Expr,BExpr,eval,evalB]
  IMPORTING simple_declarations[Vars,Value]
  CONVERSION deterministic[State].select

  SimpleProgram : TYPE = [# decl : VarDecl, body : SimpleCommand #]
```

```
  % ---------- semantics

  p : VAR SimpleProgram

  [||](p) : set[State] =
    [| body(p) |]([| decl(p) |](emptystate))

END simple_program
```

D. Proof of Lemma `while_strans_adm`

In section 4.2, an example proof by fixed-point induction is illustrated. However, we have omitted the proof that the used predicate is admissible for fixed-point induction. For completeness the proof script of the corresponding lemma is printed below. The main idea is to characterize the least upper bound of the chain `C!1` of functions of type `srel`. As such functions map states to a set of states, the least upper bound of `C!1` is the function mapping a state to the union of all images of the functions in `C!1`.

```
(""
 (AUTO-REWRITE-DEFS)
 (STOP-REWRITE "partial_order?")
 (ASSERT)
 (SKOSIMP* :PREDS? T)
 (PROP)
 (("1" (INST?))
  ("2"
   (CASE-REPLACE
    "lub(C!1)=lambda (s:sigma): predicates.\/(fset_image(C!1)(s))")
   (("1"
     (EXPAND "\/")
     (EXPAND "chain?")
     (SKOSIMP* :PREDS? T)
     (EXPAND "fset_image")
     (SKOSIMP* :PREDS? T)
     (INST? :SUBST ("x" "f!3" "y" "f!2"))
     (EXPAND "<=")
     (EXPAND "<=")
     (PROP)
     (("1" (INST - "s!1" "x!2") (INST? :SUBST ("x" "f!2")) (REDUCE))
      ("2" (INST - "s!1" "y!1") (INST? :SUBST ("x" "f!3")) (REDUCE))))
    ("2"
     (REWRITE "func_lub_is" :DIR RL :SUBST ("S" "C!1"))
     (APPLY-EXTENSIONALITY :HIDE? T)
     (REWRITE "pred_lub_is"))
    ("3" (USE "srel.lesseqp_TCC1") (PROP))))))
```