Formalizing Fixed-Point Theory in PVS*

F. Bartels, A. Dold, H. Pfeifer, F. W. von Henke, H. Rueß
Abt. Künstliche Intelligenz, Universität Ulm
89069 Ulm, Germany
{bartels,dold,pfeifer,vhenke,ruess}@ki.informatik.uni-ulm.de

Abstract

We describe an encoding of major parts of domain theory in the PVS extension of the simply-typed λ-calculus; these encodings consist of:

- Formalizations of basic structures like partial orders and complete partial orders (domains).
- Various domain constructions.
- Notions related to monotonic functions and continuous functions.
- Knaster-Tarski fixed-point theorems for monotonic and continuous functions; the proof of this theorem requires Zorn's lemma which has been derived from Hilbert's choice operator.
- Scott's fixed-point induction for admissible predicates and various variations of fixed-point induction like Park's lemma.

Altogether, these encodings form a conservative extension of the underlying PVS logic, since all developments are purely definitional.

Most of our proofs are straightforward transcriptions of textbook knowledge. The purpose of this work, however, was not to merely reproduce textbook knowledge. To the contrary, our main motivation derived from our work on fully mechanized compiler correctness proofs, which requires a full treatment of fixed-point induction in PVS; these requirements guided our selection of which elements of domain theory were formalized.

A major problem of embedding mathematical theories like domain theory lies in the fact that developing and working with those theories usually generates myriads of applicability and type-correctness conditions. Our approach to exploiting the PVS device of *judgements* to establish many applicability conditions *behind the scenes* leads to a considerable reduction in the number of the conditions that actually need to be proved.

Finally, we exemplify the application of mechanized fixed-point induction in PVS by a mechanized proof in the context of relating different semantics of imperative programming constructs.

*This paper appeared as the technical report UIB-96-10 from the Universität Ulm, Fakultät für Informatik. This research has been funded in part by the Deutsche Forschungsgemeinschaft (DFG) under project "*Verifix*"

Contents

1. Introduction

Domain theory is concerned with the existence and uniqueness of solutions of equations as canonical least fixed-points. It forms the mathematical basis of denotational semantics for programs, and is used in systems like LCF [GMW79] for reasoning about non-termination, partial functions, and infinite-valued data types such as lazy lists and streams.

In this paper we describe an encoding of major parts of domain theory in the PVS [ORSvH95] system, a specification and verification tool which bases on Church's higher-order logic (simply-typed λ-calculus). More precisely, our encodings consist of:

- Formalizations of basic structures like partial orders and complete partial orders (cpo's, domains).
- Various domain constructions like flat cpo's, discrete cpo's, predicate cpo's, or function cpo's.
- Notions related to monotonic functions and continuous functions.
- Knaster-Tarski fixed-point theorems for monotonic and continuous functions; the proof of the fixed-point theorem for monotonic functions requires Zorn's lemma, which has been derived from Hilbert's choice operator.
- Scott's fixed-point induction principle for admissible predicates and various variations of fixed-point induction like Park's lemma.

Altogether, these encodings form a conservative extension of the underlying PVS logic, since all developments are purely definitional.

Most of these encodings and proofs are straightforward transcriptions of textbook knowledge from Loeckx and Sieber [LS87], Winskel [Win93], Schmidt [Sch88], and Gunter [Gun92]. It was not our intention, however, to slavishly reproduce textbook knowledge. To the contrary, our main motivation came from the specific requirements of the *Verifix* project for constructing formal compiler correctness proofs that require the use of fixed-point induction (see, for example, [MO96, DvHPR96]); these requirements guided our selection of what parts of domain theory we formalized. We did not expect to find major bugs in the textbooks developments, since domain and fixed-point theory are well-established mathematical fields. However, in the course of developing formal proofs, we were able to detect slight generalizations of theorems found in textbooks that streamlined our proofs.

Another motivation for this work is to investigate the suitability of various devices of the PVS specification language like parameterized theories for encoding mathematical theories and the use of some distinctive features of PVS like semantic subtypes and *judgements*.

A major problem of semantically embedding mathematical theories like domain theory and fixed-point theory lies in the fact that both developing these theories and working with them usually generates myriads of applicability conditions; i.e. one must prove all the time that a certain structure is a complete partial order, a monotonic or continuous function, or an admissible predicate. In order to reduce the number of generated applicability conditions we have made heavy use of *judgements*, a feature recently introduced to PVS, that allow additional type information to be passed to the typechecker. Instantiation of a formal parameter requiring a monotonic function with a continuous function `f`, for example, causes the PVS type-checker to generate the verification condition that `f` is monotonic. Declaration of the judgement

```
JUDGEMENT Continuous SUBTYPE_OF Monotonic
```

however, causes the type-checker to suppress this verification condition, since this fact can now be deduced *behind the scenes*.

We think that the main contribution of this work is an extensive formalization of domain-theoretic concepts to support reasoning about fixed-points that other people can use readily, or accommodate and extend it to their own purposes. Moreover, other encodings may benefit from this work in the way parameterized theories and predicate subtypes are used to formalize mathematical structures, and in the way judgements are used to suppress immense numbers of verification conditions when working with the theory.

1.1 Overview

This paper is organized as follows. After comparing our encodings with work that we think is most closely related with ours, we give a brief overview in Section 2 on the PVS system and some of its distinctive features that support encoding of mathematical structures like domain or fixed-point theory. Section 3 comprises the main part of this paper, and includes descriptions of the PVS formalizations of complete partial orders, monotonicity, continuity, admissibility, various domain constructions, fixed-point theorems, and fixed-point induction. This part also contains quite a lot of PVS text, which has sometimes been slightly edited for presentation purposes, especially when describing the interaction with the prover we do not include in our presentation hypotheses and conclusions that are not needed any more to finish the proof. In Section 4 we demonstrate an application of this mechanized fixed-point theory by proving the *while*-rule of the Hoare calculus from a state transformer semantics of the *while*-statement. Finally, Section 5 contains some concluding remarks about the suitability of PVS for formalizing mathematical structures, and our encodings of domain and fixed-point theory are listed in the Appendix; the complete PVS sources and proofs are available from the first or the last author upon request.

1.2 Related Work

The work by Agerholm [Age94, Age95] and Regensburger [Reg94, Reg95] is most closely related to ours. The overall aim of their work is to combine HOL [GM93] with LCF [GMW79, Pau87] in order to take advantage of the LCF fixed-point theory for reasoning about arbitrary (continuous) functions and infinite-valued data types, and the simple type theory of HOL which supports reasoning about finite-valued data types and (higher-order) primitive recursion. Since LCF only deals with continuous functions, both

Agerholm and Regensburger only mechanize the fixed-point theorem and fixed-point induction for continuous functions.

Agerholm [Age94, Age95] describes an embedding of the LCF logic in the HOL [GM93] theorem proving system. His basic approach is to encode domains as a pair `(set[D], <=)`, consisting of a carrier set `set[D]` and and a relation `<=`, and constructions of domains by means of functions from pairs to pairs. This choice of encoding has the consequence that a new type discipline on domains has to be introduced. Continuous functions from a domain `set[D]` to a domain `set[E]`, for example, are encoded by a HOL function `f: D -> E`. Since HOL is restricted to total functions, function `f` above must be determined for elements outside `set[D]`. Agerholm [Age94, Age95] deals with these problems by providing syntactic notations for writing domains, continuous functions and admissible predicates. These are implemented by an interface and a number of syntactic-based proof functions. Altogether, Agorholm's extension of HOL constitutes an integrated system where the domain theory constructs look almost primitive to the user, and many facts are proved behind the scenes to support this view.

It seems to be more desirable, however, to prove domain-theoretic facts *once and for all* and to encode these facts as type information of the underlying system. In this way, Regensburger [Reg94, Reg95] extends the HOL object logic of ISABELLE [Pau94] with domain-theoretic notions by employing ISABELLE's *type class* mechanism. This mechanism permits abstracting developments over mathematical structures like partial orders and domains. Instead of type classes, we use the concept of *predicate subtypes* to parameterize with respect to mathematical structures; for the fact that the type system of PVS does not include Hindley-Milner style polymorphism, we employ theory parameterization in order to parameterize with respect to types. It is well beyond the scope of this paper to compare type classes with predicate subtype mechanisms; but type classes seem to be more powerful than the predicate subtype mechanism currently implemented in PVS in that they include conventient subtype relations like "every complete partial order is a partial order". On the other hand, their expressiveness is restricted, since, for example, dependencies between type class parameters can not be expressed.

2. A Brief Description of the PVS Specification Language

The purpose of this section is to provide a brief overview of PVS, and to introduce some definitions that are used in the sequel; 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. The PVS specification language builds on classical typed higher-order logic with the usual base types, `bool`, `nat` among others, and the function type constructor `[A -> B]`. The type system of PVS is augmented with *dependent types* and *abstract data types*.

Predicates in PVS are simply elements of type `bool` and `pred[D]`, for an arbitrary type `D`, is a notational convenience for the function type `[D -> bool]`. Since sets can be determined by a property, in the sense that the set has as elements precisely those which satisfy the property, the type `set[D]` is just a notational variant of `pred[D]` and it comprises all sets with elements of type `D`.

With the notation introduced so far one can easily define the (possibly infinite) union of a set of predicates over some type `D` as stated in 1.

```
\/(PP: set[pred[D]]): pred[D] =                                          1
   LAMBDA (d: D): EXISTS ( p:(PP)): p(d);
```

It is not difficult to see that above definition coincides with the least upper bound of `PP`, now interpreted as the set of sets over type `D`. In the following we also make use of computing the image `set_image(f)(A)` of function `f` with respect to a subset `A` of `D`, and the image `fset_image(ff)(x)` of a set of functions `ff` at point `x`.[1]

```
set_image(f: [D -> R])(A:set[D]): set[R] =                               2
  { y: R | EXISTS ( x: (A)): y = f(x) }

fset_image(ff: set[[D -> R]])(x: D): set[R] =
   { y: R | EXISTS ( f: (ff)): f(x) = y }
```

Notice that these definitions make use of specialized set notation and that the arguments are curried.

A distinctive feature of the PVS specification language are *predicate subtypes* `{x:A | P(x)}`. These subtypes consist of exactly those elements of type `A` satisfying predicate `P`. Predicate subtypes are used to explicitly constrain the domain 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. For the various function images in 2, for example, the following judgements proved to be most useful for our encodings.

[1] Given a predicate (or set) `p` of type `pred[D]` (or `set[D]`), the notation `(p)` is just an abbreviation for the predicate subtype `{ x: D | p(x) }`; this notational convenience is used heavily in the sequel.

```
JUDGEMENT set_image  HAS_TYPE                                      3
   [[D -> R] -> [(nonempty?[D]) -> (nonempty?[R])]]

JUDGEMENT fset_image HAS_TYPE
   [(nonempty?[[D -> R]]) -> [D -> (nonempty?[R])]]
```

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.

The theory `example` in 4, for example, is parameterized with respect to a non-empty type `D`, a binary predicate `<=` on this type, and an element `bottom` of `D`.

```
example[D: TYPE+, le:pred[[D, D]], bottom: D]: THEORY              4
 BEGIN
   ASSUMING
     is_cpo   : ASSUMPTION cpo?[D](le, bottom)
   ENDASSUMING
   ...
 END example
```

Furthermore, the semantic constraint `is_cpo` restricts possible theory instantiations to complete partial orders,[2] since, whenever a parameterized theory is instantiated, the PVS type-checking mechanism generates TCCs according to the given assumptions. Instead of using the assumption mechanism, one could restrict possible instantiations of `<=` and `bottom` by decorating them with corresponding predicate subtypes. It is not possible, however, to abstract theories with respect to a single formal parameter that can be instantiated with complete partial orders.

In the sequel we do not always state exact theory parameterization but only use informal descriptions such as "given the complete partial order `[D, <=, bottom]`, ..." for the example theory in 4. Moreover, declarations of the context are given as comments where necessary.

Finally, we sketch some characteristics of the PVS prover. Proofs in PVS are presented in a sequent calculus. The atomic commands of the PVS prover component include induction, quantifier instantiation, conditional rewriting, simplification using arithmetic and equality decision procedures and type information, and propositional simplification. The `skosimp*` command, for example, repeatedly introduces constants (of the form `x!i`) for universal-strength quantifiers, and `assert` combines rewriting with decision procedures. PVS has an LCF-like strategy language for combining inference steps into more powerful proof strategies. The defined rule `grind`, for example, combines rewriting with quantifier reasoning and propositional and arithmetic decision procedures; this strategy is also the workhorse for proving a large number of our formalization of domain theory.

[2]The predicate `cpo?` is defined in Section 3.2.

3. Formalizations

This chapter describes formalizations of complete partial orders (domains), continuous and monotonic functions, some basic domain constructions, the Knaster-Tarksi fixed-point theorem for monotonic functions, and various fixed-point induction principles. We start with some preliminary development on partial orders, since the theory of complete partial orders rests on the concept of the least upper bound of a set.

3.1 Partial Orders

Given a partial order `[D, <=]`, an element `x` of type `D` is said to be an *upper bound* of the subset `A` of `D` if `d <= x` for all `d` in `A`; `x` is said to be the *least upper bound* (lub) of `A` (in `D`), if `x` is the least element of the set of all upper bounds of `A` in `D`. The notions of upper bounds and least upper bounds are respectively formalized by the predicates `ub?(x, A)` and `lub?(x, A)` in 5. In addition, the set of upper bounds and least upper bounds of `A` are respectively collected in the subsets `UB(A)` and `LUB(A)` of `D`.

5
```
po[D: TYPE+, <=:(partial_order?[D])]: THEORY
 BEGIN
   x, y: VAR D
   A   : VAR set[D]

   ub?(x, A)     : bool   = FORALL (a: (A)): a <= x
   UB(A)         : set[D] = { x: D | ub?(x, A) };
   lub?(x, A)    : bool   = ub?(x,A) AND FORALL (y: (UB(A))): x <= y
   lub_exists?(A): bool   = EXISTS x: lub?(x,A)

   Lub_Exists : TYPE+ = (lub_exists?)

   LUB(A): set[D] = {x:D | lub?(x,A) }

   B: VAR Lub_Exists

   lub(B): (LUB(B)) = choose(LUB(B))

   JUDGEMENT lub HAS_TYPE [B: Lub_Exists -> (UB(B))]

   ...
 END po
```

The least upper bound `lub(B)` of a subset `B` of `D` with a non-empty set `LUB(B)` is obtained by choosing an arbitrary element from `LUB(B)` using Hilbert's choice-operator.

The judgement in 5 states the obvious fact that every least upper bound is also an upper bound. This judgement has to be stated explicitly, since the currently implemented judgement mechanism does not allow for judgements with free variables as in 6.

6
```
JUDGEMENT LUB(B) SUBTYPE_OF UB(B)
```

One way to circumvent this arbitrary restriction, however, is to declare judgements like the one in 6 in a separate theory parameterized by `B`.

Predicate `min?(x, A)` in 7 tests if `x` is a *minimum* of the set `A`, and the minimums of such a set are collected in `Min(A)`.

```
min?(x, A): bool = A(x) AND FORALL ( y: (A)): y <= x IMPLIES x = y        7

Min(A): set[D] = { x: D |  min?(x, A) }
```

Encodings of the corresponding notions of *lower bounds* `lb`, *greatest lower bounds* `glb`, and *maximums* `max?` are analogous.

3.2 Complete Partial Orders

This section formalizes the notions of complete partial orders (domains) and domain constructions which are important for the mathematical description of programming languages.

The concept of a chain is crucial for the definition of domains. Given a partial order `[D, <=]`, a nonempty set `S` with elements of type `D` is called a *chain* (in `D`) if the ordering relation `<=` restricted to `S` is linear.

```
S: VAR (nonempty?[D])                                                     8

chain?(S): bool = FORALL (x, y: (S)): (x <= y) OR (y <= x)

Chain : TYPE = (chain?)
```

Now we have collected all the ingredients to represent *complete partial orders* (cpo).

```
d : VAR D                                                                 9
<=: VAR (partial_order?[D])

precpo?(<=)    : bool = FORALL (C: Chain[D,<=]): lub_exists?[D,<=](C)
bottom?(<=)(d): bool = FORALL (x: D): d <= x

cpo?(<=, d)    : bool = precpo?(<=) AND bottom?(<=)(d)

pCPO: TYPE = (precpo?)
CPO : TYPE = (cpo?)
```

A partial order `[D, <=]` is a *pre-cpo* if for every chain `C` in `D` the least upper bound `lub(C)` exists. If, in addition, the type `D` has a least element `bottom` then `[D, <=, bottom]` is called a *cpo* (or *domain*). Notice that the encodings of these concepts in 9 are only parameterized by the type `D` and the semantic restrictions for pre-cpo's and cpo's are respectively parameterized by `<=` and the pair `(<=, bottom)`. This permits defining the predicate subtype `pCPO[D]` comprising all partial orders `<=` over type `D` that satisfy predicate `precpo?`

and the predicate subtype `CPO[D]` for the pairs `(<=, bottom)` for which predicate `cpo?` holds.

Given a pre-cpo `[D, <=]`, the following judgement directs the type-checker to suppress TCCs corresponding to the definition of pre-cpo's.

10
```
% D: TYPE+, <=: pCPO[D]

JUDGEMENT Chain[D, <=] SUBTYPE_OF Lub_Exists[D, <=]
```

Similarly, for a cpo `[D, <=, bottom]` one can, for example, show that empty sets have lub's, namely the `bottom` element.

11
```
% D: TYPE, <=: pCPO[D], bottom: (bottom?(<=))

JUDGEMENT (empty?[D]) SUBTYPE_OF Lub_Exists[D, <=]
```

A simple example of a cpo is the type of Booleans equipped with implication `=>` and the bottom element `FALSE`.

12
```
JUDGEMENT IMPLIES HAS_TYPE (partial_order?[bool])
JUDGEMENT IMPLIES HAS_TYPE pCPO[bool]
JUDGEMENT FALSE   HAS_TYPE (bottom?(IMPLIES))
```

Thus, the pair `(IMPLIES, FALSE)` is of type `CPO[bool]`.

Finally we want to express the fact that `CPO` is a subtype of `pCPO`. One possibility is to specify the projection `ordering` from cpo's into precpo's in 13 as an implicit coercion via the `CONVERSION` declaration.

13
```
ordering(cpo: CPO): preCPO = proj_1(cpo)

CONVERSION ordering
```

This declaration causes the PVS type-checker to implicitly coerce objects `cpo` of type `CPO` to `ordering(cpo)` whenever an object of type `preCPO` is expected. In this way, formal parameters of type `pCPO` can be instantiated with actual parameters of type `CPO`.

3.3 CPO Constructions

In the previous sections we have introduced a number of concepts of domain theory by their semantic definitions. In this section, we introduce four example constructions on cpo's, namely discrete pre-cpo's, flat cpo's, function space cpo's, and predicate cpo's.

3.3.1 Discrete pre-CPOs

For every nonempty type `D`, the pair `[D, =]` forms both a partial order and a pre-cpo, the so-called *discrete* pre-cpo for `D`. Discrete cpo's are useful for making arbitrary PVS types into pre-cpo's.

14
```
% D: TYPE+                This is a Comment!

JUDGEMENT = HAS_TYPE (partial_order?[D])

only_trivial_chains : LEMMA
  FORALL (C: Chain[D, =]): unique?(C)

JUDGEMENT = HAS_TYPE pCPO[D]
```

All chains in a discrete cpo are trivial, since the `unique?(C)` predicate from the PVS prelude holds if and only if there is at most one element of `D` in the set `C`.

3.3.2 Lifting

Using the lifting construction one can construct a domain from an arbitrary non-empty type by adding a bottom element.

15
```
flat[D: TYPE+]: DATATYPE
  BEGIN
    elem(arg: D): elem?
    bot         : bot?
  END flat

CONVERSION elem
```

Technically, we construct a polymorphic sum type `flat` in 15 as a non-recursive data type with two constructors `elem` for injecting elements of type `D` and a constructor `bot` for the added `bottom` element. The conversion declaration in 15 causes the PVS type-checker to implicitly coerce elements `d` of type `D` to `elem(d)` whenever an element of type `flat[D]` is expected. The type `flat[D]`, equipped with the partial order `<=` as defined in 16 and the constant `bot`, forms a cpo.

16
```
% D: TYPE+

<=(d1, d2: flat): bool = (d1 = d2) OR (d1 = bot)

JUDGEMENT <= HAS_TYPE (partial_order?[flat])
JUDGEMENT <= HAS_TYPE pCPO

flat_is_cpo: LEMMA cpo?(<=, bot)
```

3.3.3 Function Domains

Let D be an arbitrary nonempty type and [R, <=, bottom] be a cpo, then one can show that the function type [D -> R] equipped with the pointwise ordering also forms a cpo. In order to make these notions precise, we first define pointwise orderings on functions.

Given a type D, a partial order [R, <=], and functions f, g of type [D ->R], f is said to be *pointwisely smaller* than g if the result of applying f to some argument is always smaller (now with respect to the ordering on the codomain R) than the result of applying g to this very same argument.

```
% D, R: TYPE+, le: (partial_order?[R])                                    17

<=(f, g: [D -> R]) : bool = (FORALL (x: D): le(f(x), g(x))

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

JUDGEMENT fset_image HAS_TYPE [Chain[[D -> R], <=] -> [D -> Chain[R, le]]]
JUDGEMENT fset_image HAS_TYPE
  [Lub_Exists[[D -> R], <=] -> [D -> Lub_Exists[R, le]]]
```

Pointwise ordering on functions [D -> R] is a partial order if [R, <=] is a partial order. The last two judgements in 17 state that the fset_image as defined in 2 preserves both the chain property and the existence of least-upper bounds.

Now, given a non-empty type D and a pre-cpo [R, le], the structure [[D -> R], <=] with <= the pointwise ordering on this function space can be shown to form a pre-cpo.

```
% D, R: TYPE+, le: pCPO[R]                                                18

JUDGEMENT pointwise[D, R, le].<= HAS_TYPE pCPO[[D -> R]]
```

If, in addition, the bottom element on the function space [D -> R], denoted by abort, is taken to be the constant function, that always returns the bottom element of the co-domain cpo over type R, then, in addition, the structure [[D -> R], <=, abort] forms a cpo.

```
% D, R: TYPE+, le: pCPO[R], bottom: (bottom?(le))                         19

abort: [D -> R] = LAMBDA (x: D): bottom

JUDGEMENT abort HAS_TYPE (bottom?[D -> R])
```

3.3.4 Predicate CPOs

Predicates on some arbitrary type D are elements of type pred[D] (or set[D]).

```
<=(p, q: pred[D]): bool = FORALL (x: D): p(x) IMPLIES q(x);                20

bottom: pred[D] = (LAMBDA (x: D): FALSE)
top   : pred[D] = (LAMBDA (x: D): TRUE)

/\(p, q: pred[D]): pred[D] = (LAMBDA (x: D): p(x) AND q(x))
\/(p, q: pred[D]): pred[D] = (LAMBDA (x: D): p(x) OR q(x))
```

It is straightforward to establish that `[pred[D], <=, bottom]` with the partial order `<=` and the `bottom` element as defined above form a cpo (actually, a complete lattice).

The definition of `<=` in 20 and the proof that `[pred[D], <=]` forms a pre-cpo, however, are superfluous, since this proof can be "inherited" using theory import of more basic constructions.

More precisely, the following import defines the pointwise ordering `<=` on `pred[D]` and establishes the fact that `pred[bool]` is a partial order (see 17).

```
IMPORTING pointwise[D, bool, IMPLIES]                                      21
```

The theory import in 21 generates the TCC that `[bool, IMPLIES]` forms a partial order; this has already been shown in 12. Further theory import of the theory described in 18 provides us with the fact that `[pred[D], <=]` forms a pre-cpo. Thus, it only remains to show that the `bottom` element as defined in 20 indeed is a bottom element.

```
JUDGEMENT bottom HAS_TYPE (bottom?(pointwise[D, bool, IMPLIES].<=))        22
```

Finally, in the case of predicates, the least upper bound of a set of predicates `PP` always exists and is given by the disjunction of all the predicates in `PP` (see 1).

```
PP: VAR set[pred[sigma]]                                                   23

pred_lub: LEMMA lub(PP) = \/(PP)
```

3.4 Monotonic Functions

Let `poD` and `poR` respectively be the two partial orders `[D, <=]` and `[R, <=]`,[3] then one defines the subset of `monotonic` functions (see 24) in the usual way. Moreover, constant functions are monotonic and serve as witnesses for the nonemptyness of the space of monotonic functions. Another remarkable fact is expressed by the judgement for `set_image` in 24. It states that `set_image(f)`, for `f` monotonic, transforms chains over the domain `D` into chains over the co-domain `R`.

[3]Technically, a theory identifier like `poD` is defined in PVS by the declaration `poD: THEORY = po[D, <=]`

```
% D: TYPE+, <=: (partial_order?[D], R: TYPE, <=: (partial_order?[R])      24

monotonic?(f: [D -> R]): bool =
  FORALL (s1,s2: D): s1 <= s2 IMPLIES f(s1) <= f(s2)

const_monotonic: LEMMA
  FORALL (c: R): monotonic?(LAMBDA (x: D): c)

Monotonic: TYPE+ = (monotonic?)

JUDGEMENT monotonic? HAS_TYPE (nonempty?[[D -> R]])
JUDGEMENT set_image  HAS_TYPE [Monotonic -> [poD.Chain -> poR.Chain]]

lub_of_monotonic_func: LEMMA
   FORALL (f: Monotonic, L: poD.Lub_Exists):
     lub_exists?(set_image(f)(L)) IMPLIES
        (lub(set_image(f)(L)) <= f(lub(L)))
```

The rather technical lemma `lub_of_monotonic_func` in 24 has been included into this text, since it forms a major part of the lemma `le_pred_admissible` in 30, which is crucial in the proof of the fixed-point theorem for monotonic functions.

3.5 Continuous Functions

The subset of `continuous` functions in 25 comprises all functions, intuitively speaking, which are compatible with the construction of least upper bounds. More precisely, given two pre-cpo's `[D, le_D]`, `[R, le_R]` a function `f` with domain `D` and codomain `R` is said to be *continuous* if for every chain `C` in the partial order `D` the least upper bound of the image `f(C)` exists and if `f(lub(C)) = lub(f(C))`.

```
continuous?(f: [D -> R]): bool =                                          25
  FORALL (C: poD.Chain):
    lub_exists?(set_image(f)(C)) AND f(lub(C)) = lub(set_image(f)(C))

Continuous: TYPE+ = (continuous?)

JUDGEMENT Continuous SUBTYPE_OF Monotonic
```

The judgement permits using continuous functions whenever a monotonic function over the same domain `D` and codomain `R` is expected, and suppresses the generation of TCCs in these cases.

Given a type `D` and a pre-cpo `[R, <=]`, the set of functions from the discrete pre-cpo `[D, =]` (see 14) to the pre-cpo `[R, <=]` are continuous.

```
% D, R: TYPE+, <=: pCPO[R]                                                26

JUDGEMENT [D -> R] SUBTYPE_OF Continuous[D, =, R, <=]
```

Finally, given three pre-cpo's `[A, <=]`, `[B, <=]`, `[C, <=]` one can easily prove that function composition, as defined polymorphically in the PVS prelude, preserves continuity.

27
```
contAB: THEORY = continuous[A, <=, B, <=]
...
JUDGEMENT o HAS_TYPE
  [contAB.continuous, contBC.continuous -> contAC.continuous]
```

Here, `continuous[...]` denotes an instantiation of the theory where the subset of continuous functions is defined. An analogous judgement about function composition holds for partial orders `A`, `B`, `C` and monotonic functions.

3.6 Admissibility

Fixed-point induction (see 3.7.3) requires the concept of *admissible predicates* as defined in 28. Moreover, we use the concept of admissibility and some related facts for proving the fixed-point theorem for monotonic functions in 3.7.2.

Let `[D, <=]` be a pre-cpo and `P` be a predicate on `D`. The predicate `P` is called admissible (see 28) if for every chain `C` the least upper bound of `C` satisfies `P` whenever all elements of `C` do; admissible predicates over `D` are collected in the predicate subtype `Admissible`.

28
```
admissible?(P: pred[D]): bool =
  FORALL (C: Chain): every(P)(C) IMPLIES P(lub(C))

Admissible : TYPE = (admissible?)
```

Some sufficient conditions for admissibility are listed in 29. These kinds of theorems are used heavily for establishing admissibility of predicates, mainly in fixed-point induction proofs.

29
```
JUDGEMENT /\ HAS_TYPE [Admissible, Admissible -> Admissible]
JUDGEMENT \/ HAS_TYPE [Admissible, Admissible -> Admissible]

JUDGEMENT /\ HAS_TYPE
  [{ PP: set[pred[D]] | every(admissible?)(PP)} -> Admissible]
```

Using the first two judgements above, the type-checker is able to deduce, for example, admissibility of `P /\ (Q \/ R) /\ Q` from the admissibility of `P`, `Q`, and `R` automatically. The last judgement in 29 states that arbitrary, possibly infinite conjunctions of admissible predicates are admissible.

The lemma `continuous_admissible` implies that statements about continuity are admissible for the pointwise function cpo; here `[D, <=]` and `[R, <=]` are pre-cpo's.

```
% D: TYPE+, <=: pCPO[D], R : TYPE+, <=: pCPO[R]                    30

continuous_admissible: LEMMA
   admissible?[[D->R], pointwise.<=](continuous?)

cont_pred_admissible: LEMMA
   FORALL (f: Continuous, P: Admissible[R, <=]):
      admissible?[D, <=](LAMBDA d: P(f(d)))

le_pred_admissible: LEMMA
   FORALL ( f: Continuous, g: Monotonic):
     admissible?[D, <=](LAMBDA  d: f(d) <= g(d))
```

The last two lemmas in $\boxed{30}$ state sufficient conditions for admissibility predicates involving continuous functions. Notice that lemma `le_pred_admissible` in $\boxed{30}$ is a slight generalization – at least for the case $n = 1$ — of Theorem 4.27 on p. 85 in [LS87], since Loeckx and Sieber require both `f` and `g` to be continuous. This generalization is actually needed in our proof of the fixed-point theorem in 3.7.2.

Moreover, for a type `D` and a cpo `[D, <=, bottom]`, the `monotonic?` predicate is admissible.

```
% D: TYPE+, <=: (partial_order?[D]),                               31
% R: TYPE+, <=: pCPO[R], bottom: (bottom?[R](<=))

monotonic_admissible: LEMMA
   admissible?[[D -> R], pointwise.<=](monotonic?)
```

Using this result it is not difficult, for example, to show that the monotonic functions with pointwise ordering and the function always returning `bottom` form a cpo.

3.7 Formalization of Fixed-Point Theory

3.7.1 Fixed-Points

Let `[D, <=]` be a partial order and `f` of type `[D -> D]` be some function; one says `x` of type `D` is the *least fixed-point* of `f` if `x = f(x)` and, whenever `y = f(y)`, one has `x <= y`; the set predicate `least_fixpoint?(f)` in $\boxed{32}$ formalizes this notion and the type `LFP(f)` comprises all least fixed points of `f`. Whenever the set of least fixed-points for a function `f` is nonempty, we say that the fixed-point for this function exists.

32

```
% D: TYPE+, <=: (partial_order?[D])

x, y: VAR D;
f: VAR [D -> D]

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

least_fixpoint?(f)(x) : bool =
  fixpoint?(f)(x) AND FORALL y: fixpoint?(f)(y) IMPLIES x <= y

least_fix_unique: LEMMA unique?(least_fixpoint?(f))

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

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

lfp_singleton : COROLLARY
  FORALL (f: Mu_Exists): singleton?(least_fixpoint?(f))

mu(f: Mu_Exists): LFP(f) = choose(least_fixpoint?(f))
```

Lemma `least_fix_unique` in $\boxed{32}$ states that least fixed-points are unique, and the proof of this lemma uses the fact that `<=` is antisymmetric (since it is a partial order).

In the case of continuous functions `f` it is straightforward to characterize the least fixed-point as the least upper bound of the set obtained by repeatedly applying `f` to the least element of its domain. Here, however, we want to deal with arbitrary functions `f` for which the least fixed-point, denoted by `mu(f)`, exists. Thus, we restrict the domain of `mu` to the predicate subtype `Mu_Exists`, and the definition of `mu(f)` involves Hilbert's ϵ-operator to `choose` an arbitrary value from the (nonempty) set of least fixed-points for `f`.[4]

From the definitions in $\boxed{32}$ it is straightforward to prove that every least fixed-point of `f` is equal to `mu(f)` and the well-known fixed-point equality `f(mu(f)) = mu(f)`.

33

```
mu_rew        : LEMMA least_fixpoint?(f)(x) IMPLIES x = mu(f)
mu_is_fixpoint: LEMMA FORALL (f: Mu_Exists): f(mu(f)) = mu(f)
```

These lemmas are proved by repeatedly unfolding definitions; in addition, the proof of `mu_rew` also requires the lemma `lfp_singleton` from $\boxed{32}$.

3.7.2 Fixed-Point Theorem for Monotonic Functions

The key result for reasoning about fixed-points is the celebrated *Knaster-Tarski* fixed-point theorem. The theorem by Knaster [Kna28] applied only to power sets and Tarski [Tar55] generalized it to complete lattices. In this section, we describe a mechanized proof of the Knaster-Tarski theorem for monotonic functions on cpo's; this presentation closely follows the proof outline given in [Ber96].

[4] `choose(S: (nonempty?[D])): (S) = epsilon(S)` is defined in the PVS prelude.

If [D, <=, bottom] is a cpo then the Knaster-Tarski fixed-point theorem states that the least fixed-point exists for monotonic functions, which in our terminology reads as follows.

```
% D: TYPE+, <= : pCPO[D], bottom : (bottom?(<=))                    34

JUDGEMENT Monotonic SUBTYPE_OF Mu_Exists
```

This judgement generates a type-correctness condition, that is closer to mathematical practice.

```
FORALL (f: Monotonic): mu_exists?(f)                                35
```

The proof of the Knaster-Tarski fixed-point theorem for monotonic functions is much harder than the one for continuous functions, and involves the use of the following variant of Zorn's lemma.

```
% D: TYPE+, <=: partial_order?[D]                                   36

IMPORTING po[D, <=]

A: VAR (nonempty?[D])
C: VAR Chain

Zorns_lemma: LEMMA
  (FORALL C: subset?(C, A) IMPLIES nonempty?[D](intersection(A, UB(C))))
     IMPLIES nonempty?[D](Max(A))
```

Informally, Lemma zorn in 36 states: if every chain, restricted to elements of some nonempty set S with elements in D, has an upper bound in S then S possesses a maximal element. Zorn's lemma can be shown to be equivalent to the Axiom of Choice. Our proof of Zorn's lemma in the PVS logic, however, uses Hilbert's ϵ-operator, which is equivalent to the Axiom of Choice.[5]

Furthermore, the main notion in the following proof of the fixed-point theorem is that of f-closed sets. These sets are required to contain the bottom element, they must be admissible[6], and whenever y is in such a set then f(y) is also in this set. This leads to the definition of f-closed subsets S of D by the predicate closed?(f)(S)in 37.

```
f: VAR Monotonic; S: VAR set[D]                                     37

step_closed?(f)(S): bool = (FORALL (y: (S)): S(f(y)))

closed?(f)(S): bool =
   contains?(bottom)(S) AND step_closed?(f)(S) AND admissible?(S)
```

[5] Our encodings for the proof of Zorn's lemma are listed in Appendix G, and the complete proof of Zorn's lemma from the ϵ-operator can be obtained from the first or the last author upon request.

[6] Here, the argument to admissible? (see 28) is interpreted as a set over D.

Now we have collected all the ingredients to describe the proof of the fixed-point theorem [34] for monotonic functions. This proof defines the least fixed-point of `f` as the maximum of the smallest `f`-closed set. Following [Ber96], the proof is split intro three parts.

First, we define the smallest `f`-closed set, denoted by the definition `X(f)` in [38].

```
X(f): set[D] = /\(closed?(f))                                              38

JUDGEMENT X HAS_TYPE [Monotonic -> (contains?(bottom))]
JUDGEMENT X HAS_TYPE [f: Monotonic -> (step_closed?(f))]
JUDGEMENT X HAS_TYPE [Monotonic -> Admissible]

X_is_closed      : LEMMA closed?(f)(X(f))
X_is_least_closed: LEMMA closed?(f)(S) IMPLIES subset?(X(f), S)
```

The proofs of the TCCs corresponding to the `contains?(bottom)` and `step_closed` judgements in [38] are trivial, and admissibility of `X(f)` is proved by skolemization, unfolding of the definition `X`, and use of lemma `adm_and_inf` in Appendix E.1.[7] These steps result in the following trivial subgoal.

```
  |-------
{1}  every(admissible?[D, <=])(closed?(f!1))
```

In addition, `X_is_closed` in [38] follows directly from these judgements and `X_is_least_closed` is proved automatically (using `grind`).

```
u(f): D = choose(Max(X(f)))                                                39

JUDGEMENT u HAS_TYPE [f: Monotonic -> (Max[D, <=](X(f)))]
JUDGEMENT u HAS_TYPE [f: Monotonic -> (X(f))]
```

The least fixed-point of `f` is defined in [39] as an arbitrary maximum element of `X(f)`. For the semantic constraint on possible arguments to `choose`, `u(f)` is only well-defined if there is a maximum element in `X(f)`, and consequently, the type-checker generates the proof obligation `nonempty?[D](Max(X(f)))`. Application of Zorn's lemma (see [36]) and introduction of type information for `X(f!1)` (see [38]) yields the following subgoal:

```
{-1}    admissible?[D, <=](X(f!1))
[-2]    subset?(C!1, X(f!1))
  |-------
{1}    nonempty?[D](intersection(X(f!1), UB(C!1)))
```

Using the definition of `admissible` (see [28]), this subgoal reduces to:[8]

[7] Lemma `adm_and_inf` corresponds to the last judgement in [29].

[8] Using the fact that `every(P)(S)` is equivalent to `subset?(S, P)`.

```
{-1}    X(f!1)(lub(C!1))
[-2]    subset?(C!1, X(f!1))
  |-------
[1]    nonempty?[D](intersection(X(f!1), UB(C!1)))
```

For the first assumption, `lub(C!1)` is an element of `X(f!1)`, and we can reduce the original goal, using some more unfolds on definitions, to the following simple fact about least upper bounds.

```
  |-------
{1}    ub?(lub(C!1), C!1)
```

This concludes the proof of the applicability condition `nonempty?[D](Max(X(f)))` and, consequently, the definition of `u(f)` in 39 is well-defined; it remains to show that `u(f)` indeed is the least fixed-point of `f`. In the second part of this proof of the fixed-point theorem, it is shown that `u(f)` is a fixed-point of `f`, and the third part finishes the proof by showing that `u(f)` is the least fixed-point of `f`.

In order to show that `u(f)` is a fixed-point, one defines the set `E(f)` of `f`-expanded elements in 40, and shows that this set is `f`-closed.

```
E(f): set[D] = { x: D| x <= f(x) }                                    40

E_is_closed : LEMMA closed?(f)(E(f))
```

Obviously, `bottom` is in `E(f)` and the proof of the second condition for `f`-closedness involves monotonicity of `f`; both conditions are proved automatically with `grind`. Furthermore, admissibility follows directly from rewriting with `le_pred_admissible` (see 30) and a lemma `const_continuous` expressing the continuity of the function returning a constant value (see Appendix E.4). Notice that the latter fact is needed to establish the applicability condition of `le_pred_admissible` when instantiated with the identity function.

Since `X(f)` is the smallest `f`-closed set and `E(f)` is `f`-closed, the set `X(f)` is a subset of `E(f)`. Thus, `u(f)` is a member of `E(f)` and consequently `u(f) <= f(u(f))`. On the other hand, since `f(u(f))` is also in `X(f)`, it follows from the maximality property of `u(f)` that `f(u(f))` is not strictly larger than `u(f)`. Thus, `u(f)` is a fixed-point of `f`.

```
u_is_fixed_point: LEMMA fixpoint?(f)(u(f))                            41
```

Skolemization, introduction of type information for `u(f!1)` and `X(f!1)` reduces the lemma above to proving the following subgoal:

```
[-1]    X(f!1)(f!1(u(f!1)))
[-2]    X(f!1)(u(f!1))
[-3]    FORALL (y: (X(f!1))): u(f!1) <= y IMPLIES u(f!1) = y
  |-------
[1]    (f!1(u(f!1)) = u(f!1))
```

More specifically, introduction of type information for `u(f!1)` yields, for the judgements in 39, the hypotheses `[-2]` and `[-3]`, and introduction of type information for `X(f!1)` yields, after some trivial manipulations, the hypothesis `[-1]`. Now, instantiation of `y` in hypothesis `[-3]` with `f!1(u(f!1))`, use of the facts `E_is_closed` (see 40) and `X_is_least_closed` (see 38), instantiation of the quantifiers `y` with `f!1(u(f!1))`, and propositional reasoning leaves us to prove:

```
[-1]    subset?(X(f!1), E(f!1))
[-2]    X(f!1)(u(f!1))
  |-------
[1]    u(f!1) <= f!1(u(f!1))
```

From the definition of `E` it is immediate that this goal holds, since `u(f!1)` is an element of `X(f!1)` and `X(f!1)` is a subset of `E(f!1)`; PVS discharges this proof obligation without any further interaction.

In the third part of our proof of the fixed-point theorem it remains to show that `u(f)` is the least fixed-point of `f`. We first define the set `V(x)` of elements smaller or equal to `x`, and show that this set is `f`-closed provided `x` is a fixed-point of `f`.

42
```
V(x): set[D] = {y: D | y <= x}

V_is_closed: LEMMA fixpoint?(f)(x) IMPLIES closed?(f)(V(x))

u_is_least_fixpoint: LEMMA least_fixpoint?(f)(u(f))

JUDGEMENT u HAS_TYPE [f: Monotonic -> LFP(f)]

KnasterTarski: THEOREM
  mu_exists?(f)

JUDGEMENT Monotonic SUBTYPE_OF Mu_Exists
```

The only non-trivial part of the `f`-closedness proof of `V(x)` involves admissibility of `V(x)`. This can be shown using `le_pred_admissible` (see 30) instantiated with the identity function and the constant function, since both functions are continuous. Thus, rewriting with these facts establishes the admissibility condition for `V(x)`. Finally, lemma `u_is_least_fixpoint` in 42 requires `u(f!1) <= y!1` for an arbitrary fixed-point `y!1`. Since `V(x)` is `f`-closed (Lemma `V_is_closed` in 40) and `X(f!1)` is the smallest `f`-closed set (Lemma `V_is_least_closed`), this reduces to the trivial goal:

```
[-1]    subset?(X(f!1), V(y!1))
  |-------
[1]    u(f!1) <= y!1
```

This finishes the proof of `u_is_least_fixpoint` and, consequently, of the Knaster-Tarski fixed-point theorem. Furthermore, using lemma `mu_rew` (see 33), one concludes that the definitions for `mu(f)` in 32 and `u(f)` in 39 coincide for monotonic functions `f`.

43
```
mu_char: LEMMA mu(f) = u(f)
```

3.7.3 Fixed-Point Induction

Let [D, <=, bottom] be a cpo and P be an admissible predicate, then fixed-point induction is stated as follows.

```
f: VAR Monotonic; P: VAR Admissible                                    44

fp_induction_mono: THEOREM
  (P(bottom) AND (FORALL x: P(x) IMPLIES P(f(x))))
     IMPLIES P(mu(f))
```

From the hypotheses it is clear that the set of elements for which P holds is f-closed; i.e. closed?(f)(P) holds. Thus, using the lemmas X_is_least_closed (see 38) and mu_char (see 43) one is reduced to show:

```
{-1}    subset?(X(f!1), P!1)
  |-------
[1]    P!1(u(f!1))
```

This trivially finishes the proof, since u(f!1) is a member of X(f!1) according to the last judgement in 39; consequently, a call to the strategy grind finishes the proof.

The fixed-point induction principle can be given a somewhat shorter formulation for a common special case.

```
park: LEMMA f(x) <= x IMPLIES mu(f) <= x                               45
```

The proof of Park's Lemma follows from fixed-point induction instantiated with the predicate (LAMBDA y: y <= x), and the proof of admissibility of this predicate is analogous to the admissibility proof for establishing lemma V_is_closed in 42.

The following variant of fixed-point induction has also proved to be useful in many cases.

```
P: Var Admissible                                                      46

fp_induction_mono_le: LEMMA
  (P(bottom) AND FORALL x: P(x) AND x <= f(x) IMPLIES P(f(x)))
     IMPLIES P(mu(f))
```

It is proved by applying fp_induction_mono to the predicate P /\ E(f) and admissibility of this predicate follows from adm_and in 29, le_pred_admissible in 30, and identity_continuous in Appendix E.4.

Finally, whenever the argument function, say g, of fixed-point induction is not only monotonic but also continuous, one can prove, in the usual way, the following specialization of the fixed-point induction principle for monototonic functions.

```
% D: TYPE+, <=: pCPO[D], bottom: (bottom?(<=))                          47

g: VAR Continuous

fp_induction_cont: THEOREM
   FORALL (P: Admissible):
      (P(bottom)
       & (FORALL (i: nat): P(iterate(g, i)(bottom))
            IMPLIES P(iterate(g, i + 1)(bottom))))
      IMPLIES
        P(mu(g))
```

For a full account of fixed-points and fixed-point induction for continuous functions see the theory `fixpoints_cont` in Appendix H.1.

4. Example: Fixed-Point Induction in PVS

In the previous section, we showed how to embed a considerable fragment of domain and fixed-point theory. This embedding has been used, for example, to encode the semantics of simple imperative programming constructs based on state transitions, and to derive the well-known Hoare calculus rules [PDvHR96]. In this chapter we shall consider a simple example to illustrate the use of fixed-point induction in the PVS prover. Other mechanized fixed-point induction proofs in the context of program semantics and compiler correctness proofs are described in [PDvHR96, DvHPR96].

First, we review some notions of the mechanized semantics described in [PDvHR96]. There, the notion of state transformers `srel` provides the basis for the denotational semantics of statements for a given state type `sigma`.

```
srel: TYPE = [sigma -> set[sigma]]                                      48
```

The partial ordering on `srel` is obtained by importing the theory `pointwise` (see $\boxed{17}$). The partial ordering on the range of `srel` is set inclusion, which is itself defined by instantiating `pointwise`.

```
IMPORTING pointwise[sigma, bool, =>]                                    49
IMPORTING pointwise[sigma, set[sigma], pointwise[sigma,bool,=>].<=]
```

Notice that the theory imports above generate proof obligations corresponding to the semantic requirements on actual theory parameters of the theory `pointwise`. Thus, we have to show that both `[bool, =>]` and `[set[sigma], pointwise[sigma,bool,=>].<=]` form partial orders. The first conditions follows from $\boxed{12}$ and the second one from $\boxed{12}$ and from $\boxed{17}$.

The state transformer mapping every state to the empty set is the least element with respect to `<=` and is called `abort`.

50
```
abort : srel = LAMBDA (s:sigma): emptyset

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

Since every type `sigma` forms a discrete pre-cpo (see 14) and sets together with set inclusion are a cpo (see 22), functions of type `srel` can shown to be continuous using the results in 26. Moreover, `[srel, <=, abort]` is a cpo.

Given these definitions, one can easily define state transformers for some simple imperative programming statements; for example: in 51.

51
```
f, g, X: VAR srel
b      : VAR set[sigma]

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

PSI(b, f)  : [srel -> srel] =
  LAMBDA X: IF b THEN f ++ X ELSE skip ENDIF;

while(b, f): srel = mu(PSI(b, f));
```

It is straightforward to prove the following monotonicity result about the `while`-functional `PSI` defined above (for a proof of a related statement see [PDvHR96]).

52
```
JUDGEMENT PSI HAS_TYPE [set[sigma], srel -> Monotonic]
```

For purpose of illustration we choose the derivation of Hoare's `while` rule from the denotational semantics given in 51. Hoare-triple `|=(p, f, q)` (see 53) hold if the image of the function `f` with respect to precondition `p` is included in the postcondition `q`.

53
```
p, q: VAR pred[sigma]
f   : VAR srel

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

h: VAR srel

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

By unfolding the definition of the `while` statement in 51 and propositional reasoning, the `while` rule of the Hoare calculus in 53 is restated as the following sequent of the PVS sequent calculus.[9]

[9]Remember that proofs in PVS are presented in a sequent calculus where antecedents and succedents are respectively numbered by negative and positive numbers.

```
while_rule :

[-1]  |=(p!1 /\ b!1, h!1, p!1)
  |-------
{1}   |=(p!1, mu(PSI(b!1, h!1)), p!1 /\ NOT(b!1))
```

This `while_rule` is proved using fixed-point induction. Thus, we use the theorem `fp_induction_mono` from [44] and instantiate its formal parameter `P` with

```
LAMBDA: (F: srel): |=(p!1, F, p!1 /\ NOT(b!1))
```

Application of fixed-point induction, followed by propositional reasoning and introduction of Skolem variables, yields the 3 subgoals in [54]. Notice that the monotonicity judgement of `PSI` in [51] causes the prover to suppress a subgoal corresponding to the monotonicity of the functional `PSI(b!1, h!1)`.

```
                                                                              54
while_rule.1:
{-1}  |=(p!1 /\ b!1, h!1, p!1)
  |-------
{1}   |=(p!1, abort, p!1 /\ NOT(b!1))

while_rule.2:
{-1}  |=(p!1, x!1, p!1 /\ NOT(b!1))
[-2]  |=(p!1 /\ b!1, h!1, p!1)
  |-------
{1}   |=(p!1, (IF b!1 THEN h!1 ++ x!1 ELSE skip ENDIF), p!1 /\ NOT(b!1))

while_rule.3 (TCC):

  |-------
{1}  admissible?(LAMBDA (F: srel): |=(p!1, F, p!1 /\ NOT(b!1)))
```

Subgoals `while_rule.1` and `while_rule.2` in [54] respectively correspond to the induction base and induction step of the fixed-point induction rule; these subgoals are proved with less than 15 interactions as easy as unfolding of definitions and propositional reasoning.

Furthermore, since the conclusion `P(mu(f))` of the fixed-point induction rule in [44] is constrained to admissible predicates `P` by means of predicate subtypes, an additional subgoal, a so-called type correctness condition (TCC), `while_rule.3` is generated. The proof of admissibility requires two additional lemmas and less than 10, mostly straightforward, user interactions. The critical idea in this proof is to characterize the least upper bound of chains `C!1` as follows:

```
lub(C!1) = LAMBDA (s: sigma): \/(fset_image(C!1)(s))
```

This is possible, since the chain `C!1` is a set of set functions, and the least upper bound of the set of function set images is simply the union of these sets.

5. Conclusions

A PVS formalization of central concepts of domain theory, including complete partial orders, various domain constructions, monotonic and continuous functions, the fixed-point theorem for monotonic functions, and various fixed-point induction theorems, have been described in this paper. These encodings make heavy use of parameterized theories to encode mathematical structures and features of the PVS type system like *judgements* to suppress a multitude of verification conditions.

Since it is not possible to encode in PVS mathematical structures like cpo's directly as a type, we used the mechanism of theory parameterization to parameterize developments with respect to mathematical structures. Moreover, it is possible to represent functor-like constructions of domains by means of parameterizing theories. Although the lack of parameterizing with respect to mathematical structures as a single object does not put any insurmountable constraints in principle, in practice parameter lists of theories and instantiations tend to become unnaturally long and difficult to survey; this is especially true when extending parameterized theories with other parameterized theories (see, for example, 49.

Another characteristics of our encodings is the consequent use of predicate subtypes and judgements; this drastically simplifies proofs, since many applicability conditions are deduced behind the scenes. On the other hand, we also experienced, besides some imperfections of the current implementation, some conceptual shortcomings of the judgement mechanism in PVS. Most importantly, an extension of the current judgement mechanism that permits for free variables in judgement declarations has the potential to considerably streamline our domain and fixed-point theory encodings.[10] Furthermore, declaration of judgements like "`(abort, <=)` has type `CPO[D]`" or, even more interestingly, "`CPO[D]` is a subtype of `pCPO[D]`" are currently not possible.

In the course of this work it became evident that the modeling of mathematical structures as a *single* type leads to more natural and elegant encodings, and that the use of *behind the scene* inference mechanism lead to simplified mechanized proofs that correspond closely to the ones found in textbooks. These are exactly the kinds of features that have recently been added to the TYPELAB [vHLP+96] system. The language of TYPELAB permits representing mathematical structures as types and abstracting over these types. Furthermore, its *behind the scenes* reasoning mechanism bases on the concept of subsumption in terminological logics [SLW96], and aims at arranging mathematical entities and components such as conceptual vocabulary or parameterized specification in a taxonomy.

Although our encodings of fixed-point induction form a conservative extension (in fact, a definitional extension) of the underlying PVS theory, and consequently do not strengthen this logic, they permit natural formalization of many proofs by mixing fixed-point induction with inductions already built-in to PVS like structural induction and well-founded induction. Mixing various induction principles was needed, for example, in the correctness proof of the linearization step [DvHPR96] of a compiler; there, the overall strategy to prove linearization is by means of fixed-point induction, and the subgoal corresponding to the induction step is proved by structural induction on the construction of the abstract

[10] According to Owre [Owr96] this extension is going to be included in future versions.

data type representing basic block graphs. Other uses of this formalization of fixed-point theory are reported in [PDvHR96, DvHPR96].

So far we have restricted ourselves to only using pre-defined PVS strategies for applying fixed-point induction. It does not seem too difficult, however, to further automate fixed-point induction proofs by developing a specialized strategy that tries to automatically apply fixed-point induction, prove the predicate at hand to be admissible based on the basis of derived sufficient conditions, and to prove the remaining subgoals using a combination of other high-level proof strategies.

While our main emphasis so far has been on using this embedding on fixed-point theory for compiler correctness proofs, it is evident that this encoding can be accommodated to support reasoning about non-termination, partial functions, arbitrary recursive (computable) functions, and infinite values of recursive domains.

References

[Age94] S. Agerholm. A HOL Basis for Reasoning about Functional Programs. Brics report series, Department of Computer Science, University of Aarhus, Denmark, 1994.

[Age95] S. Agerholm. LCF Examples in HOL. *The Computer Journal*, 38(1), 1995.

[Ber96] R. Berghammer. Theoretische Grundlagen von Programmiersprachen und Programmentwicklung. Lecture Notes, 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.

[GM93] M.J.C Gordon and T.F. Melham. *Introduction to HOL : A Theorem Proving Environment for Higher-Order Logic.* Cambridge University Press, 1993.

[GMW79] M. J. Gordon, A. J. R. Milner, and C. P. Wadsworth. *Edinburgh LCF: a Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1979.

[Gun92] C.A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing Series. The MIT Press, 1992.

[Kna28] B. Knaster. Un Théorème sur les fonctions d'ensembles. *Annales de la Societeé Polonaise de Mathematique*, 6:133–134, 1928.

[LS87] J. Loeckx and K. Sieber. *The Foundations of Program Verification.* Series in Computer Science. Wiley-Teubner, second edition, 1987.

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

[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.

[Owr96] S. Owre. Personal Communication, 1996.

[Pau87] Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Number 2 in Cambride Tracts in Theoretical Computer Science. Cambridge University Press, 1987.

[Pau94] L.C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

[PDvHR96] 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.

[Reg94] F. Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF*. PhD thesis, Technische Universität München, 1994.

[Reg95] F. Regensburger. HOLCF: Higher Order Logic of Computable Functions. In T.E. Schubert, P.J. Windley, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Application (HOL95)*, Lecture Notes in Computer Science, pages 293–307. Springer-Verlag, 1995.

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

[SLW96] M. Strecker, M. Luther, and M. Wagner. Mathematical Concepts: A Type-Theoretic Approach. In *12 European Conference on Artificial Intelligence (ECAI'96), Workshop on Representation of Mathematical Knowledge*, 1996.

[Tar55] A. Tarski. A Lattice-Theoretic Fixpoint Theorem and its Applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[vHLP+96] F.W. von Henke, M. Luther, H. Pfeifer, H. Rueß, D. Schwier, M. Strecker, and M. Wagner. The TYPELAB specification and verification environment. In M. Nivat M. Wirsing, editor, *Proceedings AMAST'96*, pages 604–607. Springer LNCS 1101, 1996.

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

Appendix: PVS Source Files

A. All Theories

```
all_theories: THEORY

BEGIN

  % -- Preliminaries

   IMPORTING  misc, set_rewrite, function_notation

  % -- Partial Orders, CPOs

   IMPORTING po_rewrite, po, po_lems, po_restrict
   IMPORTING cpo_defs, precpo, cpo, precpo_lems, cpo_lems

  % -- Monotonicity, Continuity, Admissibility

   IMPORTING monotonic, continuous, admissible

   IMPORTING composition_po, composition_precpo
   IMPORTING precpo_automorphism, precpo_restrict

  % -- Domain Constructions

   IMPORTING bool_cpo, flat_cpo, discrete_cpo
   IMPORTING pointwise, function_cpo, function_precpo, dcpo_to_precpo
   IMPORTING predicate_lems, predicates, predicate_cpo
   IMPORTING monotonic_cpo

  % -- Zorn's lemma

   IMPORTING initial_segments, zorn, zorn2

  % -- Fixedpoints, Existence, Induction

   IMPORTING fixpoints, fixpoints_mono, fixpoints_cont

END all_theories
```

B. Preliminaries

Miscellaneous

```
misc[D: TYPE+]: THEORY

BEGIN

   S, S1, S2: VAR set[D]

   every_equiv_subset: LEMMA
      every(S2)(S1) = subset?(S1, S2)

   x: VAR D
```

```
  singleton?(S) : bool =            % fehlt in PVS prelude
     exists1! x: member(x, S)

  contains?(x)(S): bool = S(x)

  IMPORTING epsilons

  select(S: (singleton?)): (S) = epsilon(S)

END misc
```

Properties about Sets

```
set_rewrite[T: TYPE ]: THEORY

BEGIN

% Theory designed for rewriting with set properties

  P, Q, R: VAR set[T]
  x      : VAR T

  nonempty_rew: LEMMA P(x) IMPLIES nonempty?(P)

  union_empty1: LEMMA union(emptyset, P) = P
  union_empty2: LEMMA union(P, emptyset) = P
  union_empty3: LEMMA empty?(Q) IMPLIES union(P, Q) = P
  union_empty4: LEMMA empty?(P) IMPLIES union(P, Q) = Q

  JUDGEMENT union HAS_TYPE
     [(nonempty?[T]), set[T] -> (nonempty?[T])]

  JUDGEMENT union HAS_TYPE
     [set[T], (nonempty?[T]) -> (nonempty?[T])]

  distr_union_intersection1: LEMMA
    union(intersection(P, Q), intersection(P, R))
    = intersection(P, union(Q, R))

  distr_union_intersection2: LEMMA
    union(intersection(Q, P), intersection( P, R))
    = intersection(P, union(Q, R))

  distr_union_intersection3: LEMMA
    union(intersection(P, Q), intersection(R, P))
    = intersection(P, union( Q, R))

  distr_union_intersection4: LEMMA
    union(intersection(Q, P), intersection(R, P))
    = intersection(P, union(Q, R))

  intersection_subset1_l: LEMMA subset?(intersection(P, Q), P)
  intersection_subset1_r: LEMMA subset?(intersection(P, Q), Q)

  intersection_subset2_l: LEMMA
    subset?(P, Q) IMPLIES intersection(P, Q) = P
```

```
  intersection_subset2_r: LEMMA
    subset?(Q, P) IMPLIES intersection(P, Q) = Q

  nonempty_add: LEMMA nonempty?(add(x, P))

  S : VAR sequence[T]
  p : VAR pred[T]

  seq_to_set( S: sequence[T]): set[T] =
    { x | EXISTS (n: nat): x = S(n)}

  JUDGEMENT seq_to_set HAS_TYPE [sequence[T] -> (nonempty?[T])]

  seq_to_set_every: LEMMA
    every(p)(seq_to_set( S)) = every(p)( S)

  % -- Big Union:

  PP: VAR set[set[T]]

  union(PP): set[T] = LAMBDA x: EXISTS (P: (PP)): P(x)

  union_inf_subset: LEMMA
    member(P, PP) IMPLIES subset?(P, union(PP))

  unique_singleton: LEMMA
    FORALL (p: (nonempty?[T])):
      unique?(p) IMPLIES p = singleton[T](choose(p))

  singleton_unique: LEMMA
    unique?(singleton(x))

  JUDGEMENT singleton HAS_TYPE [T -> (unique?[T])]

  strict_subset_of_unique: LEMMA
    FORALL (P: (unique?[T])):
      strict_subset?(Q, P) IMPLIES empty?(Q)

  difference_singleton: LEMMA
    NOT(P(x)) IMPLIES
      difference(add(x, P), P) = singleton(x)

  strict_subset_elem: LEMMA
    strict_subset?(Q, P) IMPLIES
      EXISTS x: (P(x) AND NOT(Q(x)))

END set_rewrite
```

Image of Functions

```
function_notation[D, R: TYPE+]: THEORY

BEGIN

   e : VAR R
   d : VAR D
   f : VAR [D -> R]
   K : VAR set[D]
   S : VAR set[[D -> R]]
   P : VAR PRED[R]

  % -- The image of a function-set at one point:

   fset_image(S): [ D -> set[R]] =
    (LAMBDA d: { e: R | EXISTS (f: (S)): f(d) = e})

   CONVERSION fset_image

   fset_image_nonempty: LEMMA
     nonempty?[[D -> R]](S) IMPLIES nonempty?[R](S(d))

  JUDGEMENT fset_image HAS_TYPE
    [(nonempty?[[D -> R]]) -> [D -> (nonempty?[R])]]

  fset_image_elem: LEMMA
    S(f) IMPLIES S(d)(f(d))

  % -- Set Image

   set_image(f): [set[ D] -> set[ R]] =
     (LAMBDA (M: set[D]): { e: R | EXISTS (d: (M)): e = f(d)})

   CONVERSION set_image

   setimage_image: LEMMA
     set_image(f)(K) = image(f, K)

   set_image_nonempty: LEMMA
     nonempty?[D](K) IMPLIES nonempty?[R](f(K))

   JUDGEMENT set_image HAS_TYPE
     [[D -> R] -> [(nonempty?[D]) -> (nonempty?[R])]]

   set_image_elem: LEMMA K(d) IMPLIES f(K)(f(d));

   set_image_forall: LEMMA
     (FORALL (e: (set_image(f)(K))): P(e))
     IFF (FORALL (d: (K)): P(f(d)))

END function_notation
```

C. Partial Orders

Some Rewrites

```
po_rewrite[D: TYPE+, <=: (partial_order?[D])]: THEORY

BEGIN
  x, y, z: VAR D

  is_reflexive    : LEMMA x <= x
  is_antisymmetric: LEMMA x <= y AND y <= x IMPLIES x = y
  is_transitive   : LEMMA x <= y AND y <= z IMPLIES x <= z
END po_rewrite
```

Partial Orders

```
po[D: TYPE+, <=: (partial_order?[D])]: THEORY

BEGIN

  IMPORTING po_rewrite[ D, <=]

   x, y: VAR D
   A   : VAR set[D]

  % -- Upper and lower bounds

   ub?(x, A): bool = (FORALL (a: (A)): a <= x);
   lb?(x, A): bool = (FORALL (a: (A)): x <= a);

   UB(A): set[D] = { x: D | ub?(x, A) };
   LB(A): set[D] = { x: D | lb?(x, A) };

   lub?(x, A): bool = ub?(x,A) AND FORALL (y: (UB(A))): x <= y
   glb?(x, A): bool = lb?(x,A) AND FORALL (y: (LB(A))): y <= x

   lub_exists?(A): bool = EXISTS x: lub?(x,A)
   glb_exists?(A): bool = EXISTS x: glb?(x,A)

   singleton_lub       : LEMMA lub?(x, singleton(x))
   singleton_lub_exists: LEMMA lub_exists?(singleton(x))
   lub_exists_nonempty : LEMMA nonempty?[set[D]]( lub_exists?)

   % JUDGEMENT lub_exists? HAS_TYPE (nonempty?[set[D]])

   Lub_Exists : TYPE = (lub_exists?)
   Glb_Exists : TYPE = (glb_exists?)

   LUB(A) : set[D] = {x:D | lub?(x,A) }
   GLB(A) : set[D] = {x:D | glb?(x,A) }

   lub(B:Lub_Exists) : D = choose(LUB(B))
   glb(B:Glb_Exists) : D = choose(GLB(B))

   % JUDGEMENT lub HAS_TYPE [B: Lub_Exists -> (LUB(B))]
```

```
  % -- properties of lubs and glbs

   lub_unique      : LEMMA lub_exists?(A) IMPLIES unique?(LUB(A))
   lub_of_singleton: LEMMA lub(singleton(x)) = x

  % -- Maximal and minimal elements

   min?(x, A): bool = A(x) AND FORALL ( y: (A)): y <= x IMPLIES y = x
   max?(x, A): bool = A(x) AND FORALL ( y: (A)): x <= y IMPLIES y = x

   Min(A): set[D] = { x: D |  min?(x, A) }
   Max(A): set[D] = { x: D |  max?(x, A) }

  % -- Chains

   chain?(A): bool = nonempty?(A) AND
     FORALL (x, y: (A)): (x <= y) OR (y <= x)

   Chain : TYPE = (chain?)

   JUDGEMENT Chain SUBTYPE_OF (nonempty?[D])

  % -- Least Elements

   least_element?(x, A): bool = A(x) AND lb?(x, A)

   least_elem_is_min: LEMMA least_element?(x, A) IMPLIES min?(x, A)

END po
```

Lemmas on Partial Orders

```
po_lems[D: TYPE+, <=:(partial_order?[D])]: THEORY

BEGIN

  IMPORTING po[D, <=], set_rewrite[D], function_notation

  x, y, z, b : VAR D
  A, K       : VAR set[D]
  L          : VAR Lub_Exists
  S      : VAR Chain

  upper_bound_every: LEMMA ub?(x, A) = every(LAMBDA y: y <= x)(A)
  upper_bound_add  : LEMMA ub?(b, add(x, A)) IMPLIES ub?(b, A)
  upper_bound_trans: LEMMA ub?(x, A) AND x <= y IMPLIES ub?(y, A)
  lub_def          : LEMMA lub?(lub(L), L)
  lub_is_least     : LEMMA lub(L) <= x IFF ub?(x, L)
  lub_is_ub        : LEMMA FORALL (x: (L)): x <= lub(L)

  lub_exists_rew   : LEMMA lub?(b, A) IMPLIES lub_exists?(A)
  lub_rew          : LEMMA lub?(b, A) IMPLIES lub(A) = b

  union_bound      : LEMMA ub?(x, union(A, K)) IMPLIES ub?(x, A)
  lub_smaller_lub  : LEMMA ub?(b, add(x, L)) IMPLIES lub(L) <= b
```

```
lub_union_bound: LEMMA
  ub?(lub(L), A) IMPLIES lub?(lub(L), union(A, L))

lub_union_bound_exists: LEMMA
  ub?(lub(L), A) IMPLIES lub_exists?(union(A, L))

lub_union_bound_rew: LEMMA
  ub?(lub(L), A) IMPLIES lub(union(A, L)) = lub(L)

lub_add:        LEMMA x <= lub(L) IMPLIES lub?(lub(L), add(x, L))
lub_add_exists: LEMMA x <= lub(L) IMPLIES lub_exists?(add(x, L))
lub_add_rew:    LEMMA x <= lub(L) IMPLIES lub(add(x, L)) = lub(L)

singleton_chain: LEMMA chain?(singleton(x))

JUDGEMENT singleton HAS_TYPE [D -> Chain]

chain_add: LEMMA ub?(x, S) IMPLIES chain?(add(x, S))

% a different definition (which can also be found elsewhere):
% chain?( S: sequence[D]): bool = FORALL ( n: nat): S(n)<=S(n+1)
% is shown to be stronger:

seq_ascends: LEMMA
  FORALL (S: sequence[D]):
    (FORALL (n: nat): S(n) <= S(n+1)) IMPLIES ascends?(S, <=)

chain_seq: LEMMA
  FORALL (S: sequence[D]):
    ascends?(S, <=) IMPLIES chain?( seq_to_set(S))

union_chain_l: LEMMA
  FORALL (P: (nonempty?[D]), Q: set[D]):
    chain?(union( P, Q)) IMPLIES chain?(P)

union_chain_r: LEMMA
  FORALL (P: set[D], Q: (nonempty?[D])):
    chain?(union(P, Q)) IMPLIES chain?(Q)

union_chain: LEMMA
  FORALL (P, Q: (nonempty?[D])):
    chain?(union(P, Q)) IMPLIES (chain?(P) AND chain?(Q))

SS: VAR (nonempty?[set[D]])

union_chain_inf: LEMMA
 (every(chain?)(SS) AND
    FORALL (S1, S2: (SS)): subset?(S1, S2) OR subset?(S2, S1))
  IMPLIES
    chain?(union(SS))

PP: VAR set[set[D]]

union_bound2: LEMMA
  ub?(b, union(PP)) IFF FORALL (P: (PP)): ub?(b, P)
```

```
  lub_bound: LEMMA
    every(lub_exists?)(PP) IMPLIES
      (ub?(b, set_image(lub)(PP)) IFF FORALL (P: (PP)): ub?(b, P))

  lub_combine: LEMMA
    every(lub_exists?)(PP) IMPLIES
      (lub?(b, union(PP)) IFF lub?(b, set_image(lub)(PP)))

  lub_combine_rewrite: LEMMA
    FORALL PP:
      (every(lub_exists?)(PP) AND
          (lub_exists?(union(PP)) OR lub_exists?(set_image(lub)(PP))))
       IMPLIES
         lub(union(PP)) = lub(set_image(lub)(PP))

  lower_set_bound: LEMMA
    FORALL (Q, R: Lub_Exists):
      (FORALL (x: (Q)): EXISTS (y: (R)): x <= y)
       IMPLIES lub(Q) <= lub(R)

  least_element_singleton: LEMMA
    least_element?(x, singleton(x))

END po_lems
```

Restriction of Partial Orders

```
po_restrict[
  T : TYPE+,
  le: (partial_order?[T]),
  S: TYPE+ FROM T
]: THEORY

BEGIN

  <= : (partial_order?[S]) =
     LAMBDA (s1, s2: S): le(s1, s2)

  IMPORTING po_lems[T, le], po_lems[S, <=]

  subtype_chain: LEMMA
    FORALL (C: Chain[S, <=]): chain?[T, le](C)

  JUDGEMENT extend[T, S, bool, FALSE]
     HAS_TYPE [Chain[S, <=] -> Chain[T, le]]

  subtype_lub: LEMMA
    FORALL (M: set[S], l: S):
      lub?[T, le](l, M) IMPLIES lub?[S, <=](l, M)

END po_restrict
```

D. Complete Partial Orders

Basic Definitions

```
cpo_defs[D: TYPE+]: THEORY

BEGIN
   IMPORTING po

   b  : VAR D
   <= : VAR (partial_order?[D])

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

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

   pCPO: TYPE = (precpo?)

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

   CPO: TYPE = (cpo?)
END cpo_defs
```

Judgement(s) for Pre-CPOs

```
precpo[
  D:TYPE+,    (IMPORTING cpo_defs[D])
  <=: pCPO[D]
]: THEORY

BEGIN

  IMPORTING po_lems[ D, <=]

  K: VAR Chain

  chains_bound: LEMMA lub_exists?( K)

  JUDGEMENT Chain SUBTYPE_OF (lub_exists?)

END precpo
```

Judgement(s) for CPOs

```
cpo[
  D     : TYPE+,    (IMPORTING cpo_defs[D])
  le    : pCPO[D],
  bottom: D
]: THEORY

BEGIN
```

```
   ASSUMING
     bottom_def: ASSUMPTION bottom?(le)(bottom)
   ENDASSUMING

   IMPORTING precpo_lems[D, le]

   b : VAR D
   A : VAR set[D]

   is_bottom: LEMMA le(bottom, b)

   lub_of_empty_exists: LEMMA
      empty?(A) IMPLIES lub_exists?(A)

   JUDGEMENT (empty?[D]) SUBTYPE_OF Lub_Exists

END cpo
```

Lemmas on pre-CPOs

```
precpo_lems[
  D : TYPE+,  (IMPORTING cpo_defs[D])
  <=: pCPO[D]
]: THEORY

BEGIN

  IMPORTING precpo[D, <=]

  chain_union_lub: LEMMA
    FORALL (P, Q: (nonempty?[D])):
      chain?(union(P, Q)) IMPLIES
        (lub(union(P, Q)) = lub(P) OR lub(union(P, Q)) = lub(Q))

END precpo_lems
```

Lemmas on CPOs

```
cpo_lems[
  D     : TYPE+,   (IMPORTING cpo_defs[D])
  <=    : pCPO[D],
  bottom: (bottom?[D](<=))
]: THEORY

BEGIN
  IMPORTING cpo[D, <=, bottom]

  chain_union_lub: LEMMA
    FORALL (P, Q: set[D]):
     chain?( union(P, Q)) IMPLIES
      (lub(union(P, Q)) = lub(P) OR lub(union(P, Q)) = lub(Q))
END cpo_lems
```

E. Admissibility, Monotonicity, Continuity

E.1 Admissibility

```
admissible[
  D : TYPE+,   (IMPORTING cpo_defs[D])
  <=: pCPO[D]
]: THEORY

BEGIN

   IMPORTING precpo_lems[D, <=], predicates[D],
             predicate_lems[ D]

   admissible?(P: pred[D]): bool =
     FORALL (C: Chain): every(P)(C) IMPLIES P(lub(C))

   Admissible: TYPE+ = (admissible?)

   P, Q: VAR Admissible
   PP  : VAR set[pred[D]]
   x   : VAR D

   adm_and    : LEMMA admissible?(P /\ Q)
   adm_or     : LEMMA admissible?(P \/ Q)

   adm_and_inf: LEMMA
      every(admissible?)(PP) IMPLIES admissible?(/\(PP))

   JUDGEMENT /\ HAS_TYPE [Admissible, Admissible -> Admissible]
   JUDGEMENT \/ HAS_TYPE [Admissible, Admissible -> Admissible]

   JUDGEMENT /\ HAS_TYPE
     [{ PP: set[pred[D]] | every(admissible?)(PP)} -> Admissible]

END admissible
```

E.2 Monotonic Functions

```
monotonic[
  D : TYPE+, le_D : (partial_order?[D]),
  R : TYPE+, le_R : (partial_order?[R])
] : THEORY

BEGIN

  poD : THEORY = po[D, le_D]
  poR : THEORY = po[R, le_R]

  IMPORTING function_notation[D, R]
  IMPORTING po_lems[D, le_D]
  IMPORTING po_lems[R, le_R]

  s,s1,s2 : VAR D
  t       : VAR R
```

```
  f       : VAR [D -> R]

  monotonic?(f): bool =
    FORALL s1,s2: le_D(s1,s2) IMPLIES le_R(f(s1),f(s2))

  const_monotonic   : LEMMA monotonic?(LAMBDA s: t)
  monotonic_nonempty: LEMMA nonempty?( monotonic?)

  Monotonic: TYPE+ = (monotonic.monotonic?)

  % JUDGEMENT monotonic? HAS_TYPE (nonempty?[[D->R]])

  image_preserves_chains: LEMMA
     FORALL (K: poD.Chain, f: Monotonic):
       chain?( set_image(f)( K))

  % JUDGEMENT set_image HAS_TYPE
  % [Monotonic -> [poD.Chain -> poR.Chain]]

  lub_of_monotonic_func: LEMMA
     FORALL (f: Monotonic, L: poD.Lub_Exists):
       lub_exists?(set_image(f)(L)) IMPLIES
         le_R(lub(set_image(f)(L)), f(lub(L)))

END monotonic
```

E.3 Continuous Functions

```
continuous [(IMPORTING cpo_defs)
   D : TYPE+, le_D : pCPO[D],
   R : TYPE+, le_R : pCPO[R]
]: THEORY

BEGIN

   IMPORTING function_precpo[D, R, le_R], monotonic[D, le_D, R, le_R],
             precpo_lems[D, le_D], precpo_lems[R, le_R],
             admissible

   d : VAR D
   e : VAR R
   f : VAR [D -> R]
   C : VAR poD.Chain

   continuous?(f) : bool =
     FORALL C:
        lub_exists?(set_image(f)(C))
      & (f(lub(C)) = lub(set_image(f)(C)))

                              % Identity Function as witness
   const(e): [D -> R] = (LAMBDA d: e)

   const_continuous: LEMMA continuous?(LAMBDA d: e)

   continuous_nonempty: LEMMA nonempty?(continuous?)
```

```
   Continuous: TYPE+ = (continuous?)

   JUDGEMENT const HAS_TYPE [R -> Continuous]

  % -- Every Continuous Function is Monotonic

   continuity_monotonicity: LEMMA
     FORALL (f: Continuous): monotonic?(f)

   JUDGEMENT Continuous SUBTYPE_OF Monotonic

   continuous_rew: LEMMA
     FORALL (f: Continuous, C):
       f(lub(C)) = lub(set_image(f)(C))

  % -- The continuity predicate is admissible

   continuous_admissible: LEMMA
     admissible?[[D -> R], pointwise.<=](continuous?)

  % -- Admissible predicates:

   cont_pred_admissible: LEMMA
     FORALL (f: Continuous, P: Admissible[R, le_R]):
       admissible?[D, le_D](LAMBDA d: P(f(d)))

   le_pred_admissible: LEMMA
     FORALL (f: Continuous, g: Monotonic):
       admissible?[D, le_D](LAMBDA  d: le_R(f(d), g(d)))

   le(f: Continuous, g: Monotonic): pred[D] =
      LAMBDA d: le_R(f(d), g(d))

   JUDGEMENT le HAS_TYPE
      [Continuous, Monotonic -> Admissible[D, le_D]]

END continuous
```

E.4 Monotonicity, Continuity, and Admissibility Properties

Facts about Automorphisms on pre-CPOs

```
precpo_automorphism[ (IMPORTING cpo_defs)
  D  : TYPE+,
  <= : pCPO[D]
]: THEORY

BEGIN

  IMPORTING continuous[D, <=, D, <=], pointwise[D, D, <=]

  x : VAR D
  M : VAR set[D]
```

```
  identity_image: LEMMA set_image(lambda x: x)(M) = M

  identity_continuous: LEMMA
    continuous?(LAMBDA x: x)

  JUDGEMENT id[D] HAS_TYPE Continuous

END precpo_automorphism
```

Restriction of pre-CPOs

```
precpo_restrict[
  T : TYPE+,       (IMPORTING cpo_defs)
  le: pCPO[T],
  P : (nonempty?[T])
]: THEORY

BEGIN

  IMPORTING po_restrict[T, le, (P)], admissible[T, le]

  sub_precpo: LEMMA
    admissible?(P) IMPLIES precpo?[(P)](po_restrict.<=)

END precpo_restrict
```

F. Constructions

F.1 Boolesche CPO

```
bool_cpo: THEORY

BEGIN

  IMPORTING cpo_defs[bool]

  JUDGEMENT =>    HAS_TYPE (partial_order?[bool])
  JUDGEMENT =>    HAS_TYPE pCPO
  JUDGEMENT false HAS_TYPE (bottom?(=>))

  IMPORTING cpo[ bool, =>, false]

END bool_cpo
```

F.2 Discrete CPOs

```
discrete_cpo[D: TYPE+] : THEORY

BEGIN

  IMPORTING cpo_defs[D]

  x,y : VAR D;

  discrete_is_po: LEMMA partial_order?[D](=)

  JUDGEMENT = HAS_TYPE (partial_order?[D])

  IMPORTING po_lems[D, =]

  only_trivial_chains : LEMMA
    FORALL (C:Chain[D, =]): unique?(C)

  discrete_is_precpo: LEMMA precpo?[D](=)

  JUDGEMENT = HAS_TYPE pCPO[D]

  IMPORTING precpo[D, =]

END discrete_cpo
```

F.3 Flat CPOs

```
flat_cpo[D: TYPE+]: THEORY

BEGIN
   flat: DATATYPE
     BEGIN
       elem(arg: D): elem?
       bot: bot?
     END flat

   CONVERSION elem

   IMPORTING cpo_defs[flat]

   d, d1, d2: VAR flat

   flat_order(d1, d2): bool = (d1 = d2) OR bot?(d1)

   flat_is_po    : LEMMA partial_order?[flat](flat_order)
   flat_is_precpo: LEMMA precpo?(flat_order)
   flat_is_cpo   : LEMMA cpo?(flat_order, bot)

   JUDGEMENT flat_order HAS_TYPE (partial_order?[flat])
   JUDGEMENT flat_order HAS_TYPE pCPO

   IMPORTING cpo[flat_cpo.flat, flat_cpo.flat_order, flat_cpo.bot]

END flat_cpo
```

F.4 Function CPOs

Pointwise Ordering of Functions

```
pointwise[D, R: TYPE+, le: (partial_order?[R])]: THEORY

BEGIN

  IMPORTING po_lems[ R, le]
  IMPORTING function_notation[ D, R]

  d : VAR D
  e : VAR R
  f,g : VAR [D -> R]
  S   : VAR set[[D->R]]

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

  pointwise_is_po: LEMMA partial_order?[[D -> R]](<=)

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

  IMPORTING po_lems[ [D -> R], <=]

  chain_pointwise: LEMMA
    FORALL (S: Chain[[D -> R], <=]):
      chain?(fset_image(S)(d))

  JUDGEMENT fset_image HAS_TYPE
    [Chain[[D -> R],<=] -> [D -> Chain[R, le]]]

  func_lub_lem: LEMMA
    FORALL (S: Lub_Exists[[D -> R],<=]):
      FORALL d: lub?(lub(S)(d), fset_image(S)(d))

  func_lub_lem2: LEMMA
    FORALL S: (FORALL d: lub_exists?(fset_image(S)(d)))
      IMPLIES lub?(LAMBDA d: lub(fset_image(S)(d)), S)

  func_lubs: LEMMA
    lub_exists?(S) IFF (FORALL d: lub_exists?(fset_image(S)(d)))

  JUDGEMENT fset_image HAS_TYPE
    [Lub_Exists[[D -> R],<=] -> [D -> Lub_Exists[R, le]]]

  func_lub_is: LEMMA
    FORALL (S: Lub_Exists[[D -> R], <=]):
      (LAMBDA d: lub(fset_image(S)(d))) = lub(S)

END pointwise
```

Construction of Function Space pre-CPOs

```
function_precpo[
  D   : TYPE+,
  R   : TYPE+,  (IMPORTING cpo_defs[R])
  le_R: pCPO
]: THEORY

BEGIN

  IMPORTING pointwise[D, R, le_R], precpo_lems[R, le_R],
            cpo_defs[[D -> R]]

  functions_form_precpo: LEMMA precpo?(pointwise.<=)

  JUDGEMENT pointwise.<= HAS_TYPE pCPO[[D -> R]]

  IMPORTING precpo[[D -> R], <=]

END function_precpo
```

Construction of Function Space CPOs

```
function_cpo[
  D     : TYPE+,
  R     : TYPE+,  (IMPORTING cpo_defs[R])
  le_R  : pCPO[R],
  bottom: R
] : THEORY

BEGIN

ASSUMING
  bottom_def: ASSUMPTION bottom?(le_R)(bottom)
ENDASSUMING

  IMPORTING function_precpo[D, R, le_R]
  IMPORTING cpo[ R, le_R, bottom]

  bottom_func: [D -> R] = LAMBDA (s: D): bottom

  bottom_func_is_bottom: LEMMA
     bottom?(pointwise.<=)(bottom_func)

  functions_form_cpo: LEMMA cpo?(pointwise.<=, bottom_func)

  IMPORTING cpo[[D -> R], pointwise.<=, bottom_func]

END function_cpo
```

Discrete CPOs to pre-CPOs

```
dcpo_to_precpo[D, R: TYPE+, (IMPORTING precpo) leR : pCPO[R]]: THEORY

BEGIN
  IMPORTING po, discrete_cpo[D], continuous[D, =, R, leR]

  f,g   : VAR [D -> R]
  s1,s2 : VAR D

  discrete_func_continuous: LEMMA continuous?(f)

  JUDGEMENT [D -> R] SUBTYPE_OF Continuous

END dcpo_to_precpo
```

F.5 Monotonic CPOs

```
monotonic_cpo[
  D     : TYPE+,
  leD   : (partial_order?[D]),
  R     : TYPE+,               (IMPORTING cpo_defs[R])
  leR   : pCPO[R],
  bottom: R
]: THEORY

BEGIN

   ASSUMING
     bottom_def: ASSUMPTION
      FORALL (t: R): leR(bottom, t)
   ENDASSUMING

  IMPORTING monotonic[D, leD, R, leR]
  IMPORTING function_cpo[D, R, leR, bottom]
  IMPORTING cpo_defs[Monotonic]
  IMPORTING precpo_restrict[[D -> R],
                           pointwise.<=, monotonic.monotonic?]
  IMPORTING admissible[[D -> R], <=]

  monotonic_admissible: LEMMA
    admissible?[[D->R], <=](monotonic?)

  monotonic_forms_cpo: LEMMA
    cpo?[Monotonic](po_restrict.<=, bottom_func)

END monotonic_cpo
```

F.6 Predicate CPOs

Lifting of Boolean Connectives

```
predicates[D: TYPE+]: THEORY
BEGIN
  s     : VAR D
  p,q,b : VAR pred[D]; S: VAR set[D]

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

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

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

  select(p)(S): set[D] = { s: (S) | p(s) }

  every_select: LEMMA every(p)(select(p)(S))
END predicates
```

Facts about Liftings of Boolean Connectives

```
predicate_lems[D: TYPE+]: THEORY
BEGIN

  IMPORTING predicates[D]

  S   : VAR set[D]
  P, Q: VAR pred[D]

  every_select: LEMMA every(P)(select(P)(S))
  select_every: LEMMA select(P)(S) = S IFF every(P)(S)

  every_and: LEMMA
    every(P /\ Q)(S) IFF (every(P)(S) AND every(Q)(S))

  every_or : LEMMA
    every(P \/ Q)(S) IFF union(select(P)(S), select(Q)(S)) = S

END predicate_lems
```

Construction of Predicate CPOs

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

BEGIN

  % -- Booleans with implication form a cpo:

  IMPORTING cpo_defs, predicates[D], bool_cpo

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


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

  IMPORTING pointwise[D, bool, =>]

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

  IMPORTING dcpo_to_precpo[D, bool, =>]

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

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

  PP: VAR set[pred[D]]

  IMPORTING po[pred[D], <=]

  pred_lub       : LEMMA lub?(\/(PP), PP)
  pred_lub_exists: LEMMA lub_exists?(PP)
  pred_lub_is    : LEMMA lub(PP) = \/(PP)

END predicate_cpo
```

G. Zorn's Lemma

Basic Facts about Initial Segments

```
initial_segments[
  D  : TYPE+,
  <= : (partial_order?[D])
]: THEORY

BEGIN

   IMPORTING po_lems[ D, <=]

   C : VAR Chain
```

```
   x : VAR D
   A : VAR set[D]
   AA: VAR set[set[D]]

  % -- Initial Segments (uncommonly without the empty set)
  initial_segment?(C)(A): bool =
     nonempty?(A)
    & subset?(A, C)
    & FORALL (x: (A), y: (C)): y <= x IMPLIES A(y)

  iseg_is_chain: LEMMA
    initial_segment?(C)(A) IMPLIES chain?(A)

  isegs_subset: LEMMA
    FORALL (S1, S2: (initial_segment?(C))):
      subset?(S1, S2) OR subset?(S2, S1)

  iseg_of_isegs: LEMMA
    FORALL (S1, S2: (initial_segment?(C))):
      initial_segment?(S1)(S2) OR initial_segment?(S2)(S1)

  least_element_is_iseg: LEMMA
    least_element?(x, C)
      IMPLIES initial_segment?( C)( singleton( x))

  iseg_union: LEMMA
    nonempty?(AA) AND every(initial_segment?(C))(AA)
      IMPLIES initial_segment?(C)(union(AA))

  iseg_expand: LEMMA
    FORALL (S: Chain, T: (initial_segment?(S)), x: D):
      least_element?( x, difference(S, T))
        IMPLIES initial_segment?(S)( add(x, T))

  % -- Proper Initial Segments

   proper_initial_segment?(C)(A): bool =
     initial_segment?(C)(A) AND C /= A

  % JUDGEMENT (proper_initial_segment?(C)) SUBTYPE_OF (initial_segment?(C))

   proper_iseg_add: LEMMA
     ub?(x, C) AND proper_initial_segment?(add(x,C))(A)
       IMPLIES initial_segment?(C)(A)

   proper_iseg_subset: LEMMA
     proper_initial_segment?(C)(A)
       IMPLIES strict_subset?(A, C)

   proper_iseg_leaves_bound: LEMMA
     proper_initial_segment?(C)(A) IMPLIES
       EXISTS (x: (C)): (ub?(x, A) AND not(A(x)))

END initial_segments
```

Zorn's Lemma

```
zorn[D: TYPE+, <=: (partial_order?[D])]: THEORY

BEGIN

   ASSUMING

    IMPORTING po_lems[D, <=]

    C : VAR Chain

    bound_exists: ASSUMPTION nonempty?[D](UB(C))

   ENDASSUMING

   IMPORTING initial_segments[D, <=]

   x : VAR D
   A : VAR set[D]
   AA: VAR set[set[D]]

   Max(x): bool = FORALL (y: D): x <= y IMPLIES x = y

   open_chain?(A): bool =
     chain?(A) AND empty?[D](intersection(A, Max))

   Open_Chain: TYPE = (open_chain?)

   proper_iseg_no_max: LEMMA
     proper_initial_segment?(C)(A) IMPLIES open_chain?(A)

   % For PVS-versions to come:
   % JUDGEMENT (proper_initial_segment?(C)) SUBTYPE_OF Open_Chain

   S: VAR Open_Chain

   open_chain_bounded: LEMMA
     EXISTS (x: (complement(S))): ub?(x, S)

   extern_bounds(S): (nonempty?[D]) = difference(UB(S), S)

   phi(S): (extern_bounds(S)) = choose(extern_bounds(S))

   phi_is_ub       : LEMMA ub?(phi(S), S)
   phi_not_elem    : LEMMA NOT S(phi(S))
   add_phi_is_chain: LEMMA chain?(add(phi(S), S))

   p : D

   CC : set[set[D]] =
     { C: Chain | least_element?( p, C) AND
                  FORALL (T: (proper_initial_segment?(C))):
                    least_element?(phi(T), difference(C, T))}

   CC_contains_p: LEMMA CC( singleton(p))
```

```
   CC_nonempty  : LEMMA nonempty?(CC)

   % JUDGEMENT CC HAS_TYPE (nonempty?[set[D]])

   S1, S2: VAR (CC)

   CC_contains_chains: LEMMA chain?( S1)

   JUDGEMENT (CC) SUBTYPE_OF Chain

   R(S1, S2): (nonempty?[D]) =
     union(intersection(initial_segment?(S1),
                        initial_segment?(S2)))

   R_is_iseg1: LEMMA initial_segment?(S1)(R(S1, S2))
   R_is_iseg2: LEMMA initial_segment?(S2)(R(S1, S2))

   R_equals_one_arg: LEMMA
      S1 = R(S1, S2) OR S2 = R(S1, S2)

   CC_iseg: LEMMA
     FORALL (S1, S2: (CC)):
      initial_segment?(S1)(S2) OR initial_segment?(S2)(S1)

   CC_union_is_chain: LEMMA chain?(union(CC))

   U: Chain = union(CC)

   CC_members_U : LEMMA FORALL (S: (CC)): initial_segment?(U)(S)
   CC_contains_U: LEMMA member(U, CC)

   zorn_orig: LEMMA nonempty?(Max)

END zorn
```

Variant of Zorn's Lemma

```
zorn2[D: TYPE+, <=: (partial_order?[D])]: THEORY

BEGIN

  IMPORTING po[D,<=], zorn, po_restrict

  A : VAR (nonempty?[D])
  C : VAR Chain

  Zorns_lemma: LEMMA
    (FORALL C: subset?(C, A) IMPLIES nonempty?[D](intersection(A, UB(C))))
       IMPLIES nonempty?[D](Max(A))

END zorn2
```

H. Fixed-Points

H.1 Definitions related to Fixed-Points

```
fixpoints[D: TYPE+, <=: (partial_order?[D])]: THEORY

BEGIN

  IMPORTING po[D, <=], misc[D]

  x, y: VAR D
  f   : VAR [D -> D]

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

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

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

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

  least_fix_unique: LEMMA unique?(least_fixpoint?(f))

  lfp_singleton : COROLLARY
    FORALL (f: Mu_Exists): singleton?(least_fixpoint?(f))

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

  mu_exists_rew : LEMMA least_fixpoint?(f)(x) IMPLIES mu_exists?(f)
  mu_rew        : LEMMA least_fixpoint?(f)(x) IMPLIES x = mu(f)
  mu_is_fixpoint: LEMMA FORALL (f: Mu_Exists): f(mu(f)) = mu(f)

END fixpoints
```

H.2 Fixed-Points over Monotonic Functions

```
fixpoints_mono[
  D      : TYPE+,  (IMPORTING cpo_defs[D])
  <=     : pCPO[D],
  bottom : (bottom?(<=))
]: THEORY

BEGIN

  IMPORTING cpo[D, <=, bottom], precpo_automorphism[D, <=],
            fixpoints[D, <=], admissible[D, <=],
            zorn2[D, <=], misc[D]

    x, y : VAR D
    f    : VAR Monotonic
    S    : VAR set[D]
```

```
  step_closed?(f)(S): bool = (FORALL (y: (S)): S(f(y)))

  closed?(f)(S): bool =
    contains?(bottom)(S) AND step_closed?(f)(S) AND admissible?(S)

% -- Part I: definining a fixed point u

  X(f): set[D] = /\(closed?(f))

  JUDGEMENT X HAS_TYPE [Monotonic -> (contains?(bottom))]
  JUDGEMENT X HAS_TYPE [f: Monotonic -> (step_closed?(f))]
  JUDGEMENT X HAS_TYPE [Monotonic -> Admissible]

  JUDGEMENT X HAS_TYPE [Monotonic -> (nonempty?[D])]

  X_is_closed      : LEMMA closed?(f)(X(f))
  X_is_least_closed: LEMMA closed?(f)(S) IMPLIES subset?(X(f), S)

  X_has_max  : LEMMA nonempty?[D](Max(X(f)))
                                       % Uses Zorn's Lemma
  u(f): D = choose[D](Max(X(f)))

  JUDGEMENT u HAS_TYPE [f: Monotonic -> (Max[D, <=](X(f)))]
  JUDGEMENT u HAS_TYPE [f: Monotonic -> (X(f))]

 % -- Part II: u is indeed a fixed point

  E(f): set[D] = {x: D| x <= f(x)}

  E_is_closed : LEMMA closed?(f)(E(f))

  u_is_fixpoint: LEMMA fixpoint?(f)(u(f))

 % -- Part III: u is smallest fixed point

  V(x): set[D] = { y:D | y <= x }

  V_is_closed: LEMMA fixpoint?(f)(x) IMPLIES closed?(f)(V(x))

  u_is_least_fixpoint: LEMMA least_fixpoint?(f)(u(f))

  JUDGEMENT u HAS_TYPE [f: Monotonic -> LFP(f)]

  KnasterTarski: THEOREM
     mu_exists?(f)

  JUDGEMENT Monotonic SUBTYPE_OF Mu_Exists

 % -- Characterisation of Fixed Point

  mu_char: LEMMA mu(f) = u(f)

 % -- Fixed-Point Induction

  P: VAR Admissible
```

```
    fp_induction_mono: THEOREM
      (P(bottom) AND (FORALL x: P(x) IMPLIES P(f(x))))
IMPLIES P(mu(f))

   % -- Park's Lemma

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

   % -- Another Variant of Fixed-Point Induction

    E_is_admissible: LEMMA admissible?(E(f))

    fp_induction_mono_le: LEMMA
      (P(bottom)
       & (FORALL x: P(x) AND x <= f(x) IMPLIES P(f(x))))
      IMPLIES
         P(mu(f))

  END fixpoints_mono
```

H.3 Fixed-Points over Continuous Functions

```
fixpoints_cont[
   D     : TYPE+,      (IMPORTING cpo_defs[D])
   <=    : pCPO[D],
   bottom: (bottom?(<=))
]: THEORY

BEGIN

   IMPORTING cpo[ D, <=, bottom], fixpoints_mono[D, <=, bottom],
            po_lems[D, <=]

   n: VAR nat
   d: VAR D
   f: VAR Monotonic
   g: VAR Continuous

         % { x: D | EXISTS (n: nat): x = iterate(f, n)(bottom)}
   bottom_iterations(f): Chain[D, <=] =
     seq_to_set(LAMBDA n: iterate(f, n)(bottom))

   image_of_bi: LEMMA
      add(bottom, set_image(f)(bottom_iterations(f)))
      = bottom_iterations(f)

   lub_of_bi_is_fixpoint: LEMMA
     fixpoint?(g)(lub(bottom_iterations(g)))

   fixpoint_upper_bound: LEMMA
     fixpoint?(f)(d) IMPLIES ub?(d, bottom_iterations(f))

   fixpoint_theorem: THEOREM
     mu(g) = lub(bottom_iterations(g))
```

```
  % -- Fixed point induction for continuous functions

   IMPORTING admissible[D, <=]

   fp_induction_cont: THEOREM
     FORALL (P: Admissible):
        (  P(bottom)
         & (FORALL (i: nat): P(iterate(g, i)(bottom))
              IMPLIES P(iterate(g, i + 1)(bottom))))
        IMPLIES P(mu(g))

END fixpoints_cont
```