Mechanizing Domain Theory*

F. Bartels, H. Pfeifer, F. von Henke and H. Rueß
Universität Ulm, Fakultät für Informatik

Abstract. We describe an encoding of major parts of domain theory and fixed-point theory in the Pvs extension of the simply-typed λ-calculus; these formalizations comprise the encoding of mathematical structures like complete partial orders (domains), domain constructions, the Knaster-Tarski fixed-point theorem for monotonic functions, and variations of fixed-point induction. Altogether, these encodings form a conservative extension of the underlying Pvs logic. 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 devices of *predicate subtypes* and *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. We illustrate the applicability of our encodings by means of simple examples including a mechanized fixed-point induction proof in the context of relating different semantics of imperative programming constructs.

Key words: Domain Theory, Fixed-Point Theory, Mechanized Theorem Proving

1. Introduction

Domain theory forms the mathematical basis of denotational semantics for programs, and is used in systems like Lcf [7][13] for reasoning about non-termination, partial functions, and infinite-valued domains. Although Lcf can be tailored to reason about finite-valued datatypes and total functions, the resulting proofs tend to be rather unwieldy for the ubiquitous undefined (or bottom) elements. By contrast, logics and specification languages based on Church's simply-typed λ-calculus such as Hol [8] or Pvs [12] naturally support reasoning about finite-valued datatypes and total functions. Thus, it seems to be advantageous to combine the strengths of both approaches.

In this paper we describe such a combination by semantically embedding many basic concepts of domain and fixed-point theory into the Pvs specification language. More precisely, our encodings consist of:

- Formalizations of basic mathematical structures like partial orders and complete partial orders (cpos, domains).

* Supported in part by the Deutsche Forschungsgemeinschaft (DFG) under project "*Verifix*"

- Various domain constructions like flat cpos, discrete cpos, predicate cpos, function cpos, and product cpos.
- Notions related to monotonic functions, continuous functions, and admissible predicates.
- 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 variations of fixed-point induction like Park's lemma.

Altogether, these encodings form a conservative extension of the underlying Pvs logic, and they permit, for example, expressing partial functions as the least fixed-point of some functional and reasoning about this class of functions using a combination of fixed-point induction with induction schemes like structural or well-founded induction that are built into Pvs.

Most of these encodings are straightforward transcriptions of textbook knowledge from Loeckx and Sieber [11], Winskel [20], Schmidt [18], and Gunter [9]. A major problem of embedding mathematical theories like domain theory and fixed-point theory, however, lies in the fact that both developing these theories and working with them usually generates myriads of applicability conditions; i.e. one is continually concerned that a certain structure is a cpo, a monotonic or continuous function, or an admissible predicate.

In order to reduce the number of generated applicability conditions we make use of distinctive features of the Pvs specification language such as *predicate subtyping* and *judgements*. Predicate subtypes are used, for instance, for explicitly constraining domains and ranges of operations in a specification, while judgements allow additional type information to be passed to the type-checker in order to discharge, and consequently to suppress, a multitude of generated verification conditions during type-checking. The consequent use of these features thus minimizes the amount of required proof effort when working with the theory.

1.1. Overview

This paper is organized as follows. After comparing our encodings with work that we consider most closely related with ours, we give a brief overview in Section 2 on the Pvs system and some of its distinctive

features. Section 3 comprises the main part of this paper; it includes an overview of the encodings of complete partial orders, the notions of monotonicity, continuity, and admissibility, various domain constructions, fixed-point theorems, and fixed-point induction. For the sheer size of the formalizations we are forced to make a selection in this presentation, and we only include selected definitions and theorems—thereby omitting many interesting facts needed to support these theorems. Moreover, we concentrate on encoding techniques and we do not include any proofs, since—not too surprisingly—our formal proofs correspond rather directly to textbook proofs; see [3] for a more detailed description. Section 3 contains quite a lot of PVS text, which has been edited and typeset for presentation purposes. In Section 4 we demonstrate simple applications of our mechanized domain theory including a fixed-point induction proof of the *while*-rule of the Hoare calculus from a state transformer semantics of the *while*-statement. Finally, Section 5 contains some concluding remarks about our experience with formalizing mathematical structures in PVS; the complete PVS sources and proofs are available from the authors upon request.

1.2. Related Work

The work by Agerholm [1][2] and Regensburger [16][17] is most closely related to ours, since the aim of their work is to combine the characteristics and reasoning strengths of LCF with those of HOL; both encodings, however, are restricted to the simpler case of continuous functions.

Agerholm [1][2] describes an embedding of the LCF logic in the HOL [8] theorem proving system. His basic approach is to encode domains as a pair $(set[D], \leq)$, consisting of a carrier set $set[D]$ and a relation $\leq$ of type $D \to (D \to bool)$, 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 $f : D \to E$. Since HOL is restricted to total functions, function f above must be determined for elements outside $set[D]$. Agerholm [1][2] 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 specialized proof functions (tactics). Altogether, Agerholm's extension of HOL constitutes an integrated system where the domain theory constructs look (almost) primitive to the user, and many facts are proved, using specialized tactics, 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 [16][17] extends the HOL object logic of ISABELLE [14] with domain-theoretic notions by employing ISABELLE's *type class* mechanism, which permits a fine-grained use of polymorphism. On the other hand, the expressiveness of type classes is restricted since dependencies between type class parameters can not be expressed. Instead of type classes, we use the concepts of *predicate subtyping* to parameterize with respect to certain classes of mathematical structures. In addition, predicate subtyping supports the encoding of facts that rely on dependencies between source and target types as type information.

2. A Brief Description of the PVS Specification Language

The PVS system combines an expressive specification language with an interactive proof checker; see [12] for an overview. This section provides a brief description of the PVS language and prover, and introduces some of the concepts needed in this paper. More details can be found in [5].

The PVS specification language builds on classical typed higher-order logic with the usual base types, *bool*, *nat*, *int*, among others, the function type constructor $[D \to R]$, and the product type constructor $[A, B]$. The type system of PVS is augmented with *dependent types* and *abstract data types*. In PVS, predicates over some type A are, as usual, boolean-valued functions on A, and $pred[A]$ is an abbreviation for the function type $[A \to bool]$.

A distinctive feature of the PVS specification language are *predicate subtypes*: the subtype $\{x : D \mid P(x)\}$ consists of exactly those elements of type D satisfying predicate P. The expression (P) is an abbreviation for the predicate subtype $\{x : D \mid P(x)\}$. Since sets can be described by their characteristic predicates, the expression $\{x : D \mid P(x)\}$ can also be used to denote the set of elements satisfying P, and consequently $set[A]$ is just a notational variant of $pred[A]$. Hence, $x : (P)$ also means that x is an element of the *set* P. This feature is used, for example, in computing the image of a function f restricted to S.

$$image(f : [D \to R])(S : set[D]) : set[R] = \{ y : R \mid \exists(x : (S)) : y = f(x) \}$$

Predicate subtypes are used for explicitly constraining domains and ranges of operations in a specification and to define partial functions.

In general, type-checking with predicate subtypes is undecidable; the type-checker generates proof obligations, so-called *type correctness conditions* (TCCs) if satisfaction of the restricitng predicate 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 type-checker 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* SUBTYPE_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 type-checker 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 need not be generated.

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.

As an example, Figure 1 shows parts of the theory *po* that deals with notions related to partial orders. The theory is parameterized by a nonempty type D, since theory parameterization is the only means in PVS to parameterize with respect to types. The theory defines the type $PO[D]$ of partial orders (over D). Predicates *ub*? and *lub*? specify upper bounds and least upper bounds, respectively, of a set A, while predicate *lub_exists*? holds for sets that have least upper bounds. However, one can show that if such least upper bounds do exist, then there is at most one; this is expressed by the lemma *lub_unique*. Variables that occur free in formulae, like A in *lub_unique*, are implicitly universally quantified. Using the property *lub_unique*, one can define a function *lub* that returns the least upper bound of a set. Note that *lub* is only defined on those sets B for which it is known that a least upper bound exists. The operator *choose* is pre-defined in the PVS prelude (as is the predicate *unique*?) and takes as argument a nonempty set and returns an arbitrary element of that set. Technically, the non-deterministic *choose* function is defined from Hilbert's *epsilon*-operator.

$$choose(S : (nonempty?[D])) : (S) = epsilon(S)$$

Since we have shown that the set $LUB(B)$ is a singleton set the *choose* operator is deterministic in this case and thus *lub* is well-defined. Final-

```
po[D : TYPE+ ] : THEORY
BEGIN
    PO : TYPE+ = (partial_order?[D])

    ≤: VAR PO; x, y : VAR D; A : VAR set[D]

    ub?(≤)(x, A) : bool    = ∀(a : (A)) : a ≤ x
    UB(≤)(A)     : set[D]  = {x : D | ub?(≤)(x, A)}

    lub?(≤)(x, A) : bool =
        ub?(≤)(x, A) ∧ ∀(y : (UB(≤)(A))) : x ≤ y

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

    lub_exists?(≤)(A) : bool = nonempty?(LUB(≤)(A))

    lub_unique : LEMMA
        unique?(LUB(≤)(A))

    lub(≤)(B : (lub_exists?(≤))) : D = choose(LUB(≤)(B))

    chain?(≤)(A) : bool =
        nonempty?(A) ∧ ∀(x, y : (A)) : (x ≤ y) ∨ (y ≤ x)

    Chain(≤) : TYPE+ = (chain?(≤))
END po
```

Figure 1. Theory of Partial Orders

ly, the theory *po* specifies a predicate subtype of sets that have the chain property, i. e. there are no elements in the set that are incomparable with respect to the ordering relation $\leq$; families of types are defined using free variables as in the definition of $Chain(\leq)$.

A theory can use the definitions and theorems of another theory by *importing* it. Parameterized theories can be imported in either of two ways: first, one instantiates the theory by providing actual values for the formal parameters, or, second, the theory is imported without any instantiation. In the latter case all possible instantiations of the imported theory may be used; in case of ambiguities, actual values can be provided by qualifying on imported identifiers (as e.g. in $set[D]$).

We close our description of PVS with a brief sketch of some characteristics of the 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 the workhorse for proving a large number of our formalization of domain theory.

3. Formalization of Domain Theory

This chapter describes formalizations of complete partial orders (domains), continuous and monotonic functions, some basic domain constructions, the Knaster-Tarski fixed-point theorem for monotonic functions, and various fixed-point induction principles.

3.1. Complete Partial Orders

A partial order $\leq$ over D is a *pre-cpo* over D if for every chain C in D the least upper bound $lub(\leq)(C)$ exists. If, in addition, the type D has a least element *bottom* then the pair $(\leq,\ bottom)$ is called a *complete-partial order*. The encodings of these concepts in Definition 3.1 are packaged in a theory—called *domains*—that is parameterized by the (nonempty) type D; For sake of conciseness, however, we omit the PVS-specific syntax for denoting theories and theory parameterization.

Definition 3.1 (Domains). Let $D : \text{TYPE}^+$; then:

$$\leq : \text{VAR } PO[D];\ \ x : \text{VAR } D$$

$$precpo?(\leq) : \ bool \ = \ \forall(C : \ Chain(\leq)) : \ lub\_exists?(\leq)(C)$$

$$preCPO : \ \text{TYPE}^+ \ = \ (precpo?)$$

$$bottom?(\leq)(x) : \ bool \ = \ \forall(y : \ D) : \ x \leq y$$

$$Bottom(\leq) : \ \text{TYPE}^+ \ = \ (bottom?(\leq))$$

$$cpo?(\leq,\ x) : \ bool \ = \ precpo?(\leq) \wedge bottom?(\leq)(x)$$

$$CPO : \ \text{TYPE}^+ \ = \ (cpo?)$$

The defining predicates for pre-cpos and cpos are parameterized by $\leq$ and the pair $(\leq,\ bottom)$, respectively. This permits defining the predicate subtype $preCPO[D]$ comprising all partial orders $\leq$ over type D that satisfy predicate *precpo*?, and the predicate subtype $CPO[D]$ for the pairs $(\leq, bottom)$ for which predicate *cpo*? holds.

Alternatively, the PVS subtyping mechanism permits encoding of mathematical structures like pre-cpos by pairs $(S, \leq)$, where S is a set with elements of type D and the type of the ordering predicate $\leq$, namely $pred[[(S), (S)]]$, uses the predicate subtype (S) formed from the set S in order to restrict $\leq$ to elements of S.

$$preCPO_{alt} : \ \text{TYPE}^+ \ =$$
$$[S : \ set[D],\ \leq : \ \{\leq : \ pred[[(S), (S)]] \ \mid \ precpo?[(S)](\leq)\}]$$

Compared with *preCPO* from Definition 3.1, the encoding $preCPO_{alt}$ seems to be closer to mathematical practice, since a pre-cpo is usually thought of as a carrier set with an associated ordering. In our experience, however, switching between sets S and corresponding predicate subtypes (S) caused proofs to get rather cumbersome; consequently, we abandoned using the encoding $preCPO_{alt}$.

Simple examples of pre-cpos over a given type D are the discrete pre-cpos, and the type of Booleans *bool* equipped with implication $\Rightarrow$ and the bottom element false.[1]

$$discrete : \ preCPO[D] \ = \ (=)$$

$$bool : \ CPO[bool] \ = \ (\Rightarrow,\ \text{false})$$

[1] PVS distinguishes between the type *bool* and the cpo *bool*, because there are separate name spaces for types and for constants.

Next, we want to express the fact that $CPO[D]$ is a subtype of $preCPO[D]$. One possibility is to specify a projection function $\leq$ from cpos into pre-cpos as an implicit coercion via the CONVERSION declaration.

Definition 3.2 (Selectors). Let $D : \text{TYPE}^+$; then:

$$\downarrow_{\leq}(d : CPO[D]) : preCPO[D] = proj\_1(d)$$
$$\downarrow_{\perp}(d : CPO[D]) : Bottom(\downarrow_{\leq}(d)) = proj\_2(d)$$

$$\text{CONVERSION} \downarrow_{\leq} : [CPO[D] \to preCPO[D]]$$

This declaration causes the PVS type-checker to implicitly coerce objects d of type $CPO[D]$ to $\downarrow_{\leq}(d)$, whenever an object of type $preCPO[D]$ is expected. In this way, formal parameters of type $preCPO[D]$ can be instantiated with actual parameters of type $CPO[D]$.

3.2. CPO CONSTRUCTIONS

We exemplify the encoding of basic domain constructions by presenting the constructions of flat cpos and function space cpos. Other constructions, like product or sum cpos, are defined in a similar way.

3.2.1. *Lifting*

Using the lifting construction one constructs a domain from an arbitrary non-empty type by adding a bottom element. Technically, we construct in Definition 3.3 a polymorphic sum type *flat* as a non-recursive data type with two constructors: *inject* for injecting elements of type D and *bottom* for the added bottom element. The conversion declaration in Definition 3.3 causes the PVS type-checker to implicitly coerce elements x of type D to $inject(x)$ whenever an element of type $flat[D]$ is expected.

Definition 3.3 (Lifting). Let $D : \text{TYPE}^+$; then:

$$
\begin{array}{l}
\mathit{flat} : \text{DATATYPE} \\
\quad \text{BEGIN} \\
\quad\ \mathit{inject}(\mathit{arg} : D) : \mathit{inject}? \\
\quad\ \mathit{bottom} : \mathit{bottom}? \\
\quad \text{END } \mathit{flat}
\end{array}
$$

$$\text{CONVERSION } \mathit{inject}$$

$$\leq : \mathit{preCPO}[\mathit{flat}] = \lambda(x, y : \mathit{flat}) : (x = y) \vee \mathit{bottom}?(x)$$

$$\mathit{flat} : \mathit{CPO}[\mathit{flat}] = (\leq, \mathit{bottom})$$

The type $\mathit{flat}[D]$, equipped with the pre-cpo $\leq$ as defined in Definition 3.3 and the constant *bottom*, forms a cpo.

3.2.2. *Function Domains*

Given types D and R, the domain constructor $\mapsto$ generates a cpo over function type $[D \to R]$ from a pre-cpo over D and a cpo over R by ordering functions pointwise and using the constant bottom function.[2]

Definition 3.4 (Function Space). Let $D, R : \text{TYPE}^+$; then:

$$
\begin{array}{l}
\mathit{pointwise}(\leq : \mathit{preCPO}[R]) : \mathit{preCPO}[[D \to R]] = \\
\quad \lambda(f, g : [D \to R]) : \forall(x : D) : f(x) \leq g(x)
\end{array}
$$

$$
\begin{array}{l}
\mathit{fbottom}(\mathit{cod} : \mathit{CPO}[R]) : \mathit{Bottom}[[D \to R]](\mathit{pointwise}(\mathit{cod})) = \\
\quad \text{LET } (\leq, \mathit{bottom}) = \mathit{cod} \text{ IN} \\
\quad\quad \lambda(x : D) : \mathit{bottom}
\end{array}
$$

$$
\begin{array}{l}
\mapsto (\mathit{dom} : \mathit{preCPO}[D], \mathit{cod} : \mathit{CPO}[R]) : \mathit{CPO}[[D \to R]] = \\
\quad \text{LET } (\leq, \mathit{bottom}) = \mathit{cod} \text{ IN} \\
\quad\quad (\mathit{pointwise}(\leq), \mathit{fbottom}(\mathit{cod}))
\end{array}
$$

Now, it is easy to define the cpo of predicates over some type D and the cpo of (non-deterministic) state transformers *srel* with domain D and codomain R.[3]

[2] In general, no restrictions on the domain type are required for constructing a function space cpo; the restriction of the domain to pre-cpos, however, has the advantage of providing a hook for further subtyping constraints on the domain type—like the one in Example 3.8. On the other hand, this restriction does not pose any real limitations, since any type D can be viewed as a discrete pre-cpo.

[3] Remember, both $\mathit{pred}[D]$ and $\mathit{set}[D]$ are synonymous for $D \to \mathit{bool}$.

Example 3.5.

$$predicates : CPO[pred[D]] = (discrete[D] \mapsto bool)$$

$$srel : CPO[[D \to set[D]]] = (discrete[D] \mapsto predicates[D])$$

Using the judgement of Definition 3.4, the type-checker deduces that predicates and state transformers are continuous, and consequently suppresses corresponding TCCs.

3.3. Monotonicity, Continuity, and Admissibility

Given two partial orders over some domain type D and range type R, monotonicity is defined in the usual way.

Definition 3.6 (Monotonicity). Let $D, R : \text{TYPE}^+$; then:

$\leq_D$: VAR $PO[D]$
$\leq_R$: VAR $PO[R]$

$$monotonic?(\leq_D, \leq_R)(f : [D \to R]) : bool =$$
$$\forall(x, y : D) : x \leq_D y \Rightarrow f(x) \leq_R f(y)$$

$$Monotonic(\leq_D, \leq_R) : \text{TYPE}^+ = (monotonic?(\leq_D, \leq_R))$$

The subtype of *continuous* functions in Definition 3.7 comprises all functions, intuitively speaking, which are compatible with the construction of least upper bounds.

Definition 3.7 (Continuity). Let $D, R : \text{TYPE}^+$; then:

$\leq_D$: VAR $preCPO[D]$
$\leq_R$: VAR $preCPO[R]$

$$continuous?(\leq_D, \leq_R)(f : [D \to R]) : bool =$$
$$\forall(C : Chain(\leq_D)) :$$
$$lub\_exists?(\leq_R)(image(f)(C))$$
$$\wedge\, f(lub(\leq_D)(C)) = lub(\leq_R)(image(f)(C))$$

$$Continuous(\leq_D, \leq_R) : \text{TYPE}^+ = (continuous?(\leq_D, \leq_R))$$

The following subtype judgement expresses the fact that, given the precpos $\leq_D$ and $\leq_R$ over D and R respectively, every continuous function

from D to R is also monotonic.[4]

JUDGEMENT $Continuous(\leq_D, \leq_R)$
SUBTYPE_OF $Monotonic(\leq_D, \leq_R)$

Furthermore, given the (singleton) type $discrete\_preCPO[D]$, the judgement in Example 3.8 refines, using dependent typing, the type of the function space constructor *map* (see Definition 3.4). Informally, this judgement expresses the fact that functions from pre-cpos into cpos are continuous.

Example 3.8. Let $D, R : \text{TYPE}^+$; then:

$discrete\_preCPO : \text{TYPE}^+ = \{\leq : preCPO[D] \mid \leq = (=)\}$
JUDGEMENT $\mapsto$ HAS_TYPE
$[dom : discrete\_preCPO[D],\ cod : CPO[R]$
$\rightarrow CPO[Continuous(dom, \downarrow_{\leq}(cod))]]$

Fixed-point induction requires the concept of *admissible predicates*. Let $\leq$ be a pre-cpo and P be a predicate on D. The predicate P is called admissible if for every chain C the least upper bound of C satisfies P whenever all elements of C do; admissible predicates on D (with respect to $\leq$) are characterized by the predicate subtype $Admissible(\leq)$.

Definition 3.9 (Admissibility). Let $D : \text{TYPE}^+$; then:

$\leq$: VAR $preCPO[D]$

$admissible?(\leq)(P : pred[D]) : bool =$
$\forall(C : Chain(\leq)) : every(P)(C) \Rightarrow P(lub(\leq)(C))$

$Admissible(\leq) : \text{TYPE}^+ = (admissible?(\leq))$

Many sufficient conditions for admissibility, used mainly in fixed-point induction proofs, can be stated easily as judgements.

JUDGEMENT $\wedge$ HAS_TYPE
$[Admissible(\leq), Admissible(\leq) \rightarrow Admissible(\leq)]$

Using this judgements and a similar one for disjunction, the type-checker is able to deduce automatically admissibility of formulas like $P \wedge (Q \vee R) \wedge Q$ from the admissibility of P, Q, and R.

[4] This judgement must be expressed in a separate theory parameterized by pre-cpos $\leq_D$ and $\leq_R$, since, at least currently, PVS restricts judgements to statements not containing free variables.

Lemma 3.10.

$\leq_D$: VAR $preCPO[D]$; $\leq_R$: VAR $preCPO[R]$

$cont\_pred\_admissible$: LEMMA
$\forall(f : Continuous(\leq_D, \leq_R),\ P : Admissible(\leq_R)) :$
$admissible?(\leq_D)(\lambda\, d :\ P(f(d)))$

$le\_pred\_admissible$: LEMMA
$\forall(f :\ Continuous(\leq_D, \leq_R),\ g :\ Monotonic(\leq_D, \leq_R)) :$
$admissible?(\leq_D)(\lambda(x : D) :\ f(x) \leq_R g(x))$

Lemma *cont_pred_admissible*, for example, states a sufficient condition for admissibility predicates involving continuous functions, and lemma *le_pred_admissible* forms a crucial part in our proof of the Knaster-Tarski theorem for monotonic functions.

3.4. Formalization of Fixed-Point Theory

For the sake of completeness, Definition 3.11 formalizes, for a given type D, standard notions regarding fixed-points and least fixed points; the set $lfp?(f)$ and the corresponding type $LFP(f)$, for example, comprise all least fixed points of function f.

Definition 3.11 (Fixed-points). Let $D : \text{TYPE}^+$; then:

$\leq$: VAR $PO[D]$; $x,\ y$: VAR D; f : VAR $[D \to D]$

$fixedpoint?(f)(x) :\ bool\ =\ (f(x)\ =\ x)$

$lfp?(\leq)(f)(x) :\ bool\ =$
$fixedpoint?(f)(x) \wedge (\forall(y :\ D) :\ fixedpoint?(f)(y) \Rightarrow x \leq y)$

$lfp\_exists?(\leq)(f) :\ bool\ =\ nonempty?(lfp?(\leq)(f))$

$LFP(\leq,\ f)\ \ :\ \text{TYPE}^+\ =\ (lfp?(\leq)(f))$
$LFP\_Exists(\leq) :\ \text{TYPE}^+\ =\ (lfp\_exists?(\leq))$

The key result for reasoning about fixed-points is the celebrated *Knaster-Tarski* fixed-point theorem.[5] Given a cpo d (over type D), this

[5] The theorem by Knaster [10] applied only to power sets and Tarski [19] generalized it to complete lattices.

theorem states that the least fixed-point exists for monotonic functions over d, which in our terminology reads as follows.

JUDGEMENT $Monotonic(\downarrow_{\leq}(d), \downarrow_{\leq}(d))$
SUBTYPE_OF $LFP\_Exists(\downarrow_{\leq}(d))$

This judgement generates a type-correctness condition that states the Knaster-Tarski theorem in a form that is closer to mathematical practice.

Theorem 3.12.

$KnasterTarski$: THEOREM
$\forall(d : CPO[D],\ f : Monotonic(\downarrow_{\leq}(d), \downarrow_{\leq}(d))) :$
$lfp\_exists?(\downarrow_{\leq}(d))(f)$

The main characteristics of our mechanized proof lies in the systematic use of judgements to keep the formal reasoning as close as possible to the paper proof in [4]; see [3] for a detailed description. Most interestingly, our proof uses a variant of Zorn's lemma and, consequently, we have developed a formal proof of Zorn's lemma from Hilbert's *epsilon*-operator. Since this choice operator is included in the underlying PVS logic, we may claim that our formalization of fixed point theory forms a definitional extensional of the PVS logic; this implies, in particular, that our encodings form a conservative extension of PVS.

Given the Knaster-Tarski theorem, it is straightforward to define a least fixed-point for monotonic functions over a cpo by selecting an arbitrary element from the set of least fixed points for such a function, since Knaster-Tarski guarantees that this set is non-empty as required by the definition of *choose* (see Section 1.1).

Definition 3.13.

$\mu(d : CPO[D])(f : Monotonic(\downarrow_{\leq}(d), \downarrow_{\leq}(d))) : LFP(\downarrow_{\leq}(d),\ f) =$
$choose(lfp?(\downarrow_{\leq}(d))(f))$

Since $\mu(d)(f)$ is, by definition, a least fixed-point, one gets immediately the fixed-point equality for μ.

Lemma 3.14.

μ_char : LEMMA
$\forall(d : CPO[D],\ f : Monotonic(\downarrow_{\leq}(d),\ \downarrow_{\leq}(d))) :$
$\mu(d)(f) = f(\mu(d)(f))$

Moreover, fixed-point induction is derived from the Knaster-Tarski theorem in the usual way.

Theorem 3.15. Let d : VAR $CPO[D]$

$$
\begin{array}{l}
\mathit{fp\_induction\_mono} : \textsc{Theorem} \\
\quad \textsc{Let}\ (\leq, \mathit{bottom}) = d\ \textsc{In} \\
\quad\quad \forall(f : \ \mathit{Monotonic}(\leq, \leq), P : \ \mathit{Admissible}(\leq)) : \\
\quad\quad\quad (P(\mathit{bottom}) \wedge (\forall(x : \ D) : P(x) \Rightarrow P(f(x)))) \\
\quad\quad\quad\quad \Rightarrow P(\mu(d)(f))
\end{array}
$$

4. Examples

In the previous section, we showed how to embed a considerable fragment of domain theory. These formalizations are collected in a non-parametric, top-level theory called *domain_theory*. By including the statement

IMPORTING *domain_theory*

in a specification, the current context is extended by theses notions. Now, we shall consider examples that illustrate simple applications of the theory built up in the preceeding section for defining partial functions, closures of relations, and using the PVS prover for reasoning about partial functions using fixed-point induction.

4.1. EXAMPLE: PARTIAL FUNCTION

Assuming a suitable strict extension $* : [\mathit{flat}[\mathit{int}], \mathit{flat}[\mathit{int}] \to \mathit{flat}[\mathit{nat}]]$ of multiplication, one may, for example, define the factorial function *fac* on the integers as a partial function that yields *bottom* only on the negative integers.

Definition 4.1.

$$
\begin{array}{l}
\mathit{fac} : \ [\mathit{int} \to \mathit{flat}[\mathit{int}]] \ = \\
\quad \mu(\mathit{discrete}[\mathit{int}] \mapsto \mathit{flat}[\mathit{int}]) \\
\quad\quad (\lambda(f : \ [\mathit{int} \to \mathit{flat}[\mathit{int}]]) : \lambda(n : \ \mathit{int}) : \\
\quad\quad\quad \textsc{If}\ n = 0\ \textsc{Then}\ 1\ \textsc{Else}\ n * f(n-1)\ \textsc{Endif}\ )
\end{array}
$$

The conversion *inject* of Definition 3.3 causes the type-checker to automatically inject the THEN -part and the first argument of $*$ into the

corresponding *flat* datatype. Moreover, the monotonicity TCC generated for the application of the least fixed-point operator μ to given functional is easily discharged using the proof strategy `grind`.

4.2. Example: Transitive Closure

Given a binary relation R over some type D, one defines the transitive closure of R as the least fixed-point of the functional $(\lambda X : R \cup (X \circ X))$, where union $\cup$ and composition $\circ$ of relations are defined in the obvious way. Again, the monotonicity of this functional is proved using the `grind` strategy.

Definition 4.2. Let $D : \text{TYPE}^+$; then:

$$BinRel : \text{TYPE}^+ = pred[[D, D]]$$

$$R,\ S,\ X : \text{VAR } BinRel$$

$$\subseteq : preCPO[BinRel] = \downarrow_{\leq}(predicates[[D, D]])$$

$$\tau(R) : Monotonic(\subseteq, \subseteq) = \lambda X : (R \cup (X \circ X))$$

$$tc(R) : BinRel = \mu(predicates[[D, D]])(\tau(R))$$

It is a simple exercise to show that $tc(R)$ is indeed the transitive closure of R.

4.3. Example: Mechanized Semantics

The embedding of domain theory and fixed-point theory has been used to encode the semantics of simple imperative programming constructs based on state transitions, and to derive the well-known Hoare calculus rules [15]. First, given the state transformer type $srel : \text{TYPE}^+ = [D \to set[D]]$, we define some state transformers for some imperative programming statements.[6]

[6] Mixfix operations like IF ... THEN ... ELSE ... ENDIF can be overloaded.

Definition 4.3 (Statements). Let $D : \text{TYPE}^+$; then:

$$
\begin{array}{l}
f,\ g,\ X : \text{VAR } srel[D] \\
s \quad\quad : \text{VAR } D \\
b \quad\quad : \text{VAR } set[D]
\end{array}
$$

$$skip : srel[D] = \lambda\, s : singleton(s)$$

$$;(f,\ g) : srel[D] = \lambda\ s : image(g)(f(s))$$

$$
\begin{array}{l}
\text{IF } (b,\ f,\ g) : srel[D] = \\
\quad \lambda\, s : \text{IF } b(s) \text{ THEN } f(s) \text{ ELSE } g(s) \text{ ENDIF}
\end{array}
$$

$$\sqsubseteq : preCPO[srel[D]] = \downarrow_{\leq}(srel[D])$$

$$abort : Bottom(\sqsubseteq) = \downarrow_{\bot}(srel[D])$$

$$
\begin{array}{l}
\Psi(b,\ f) : Monotonic(\sqsubseteq, \sqsubseteq) = \\
\quad \lambda X : \text{IF } b \text{ THEN } f; X \text{ ELSE } skip \text{ ENDIF}
\end{array}
$$

$$while(b,\ f) : srel[D] = \mu(srel[D])(\Psi(b,\ f))$$

A mechanized proof of the monotonicity of functional Ψ is described in [15]. Furthermore, $srel[D]$ is overloaded, since at certain positions it denotes a type and at others the state transformer cpo defined in Example 3.5.

For purpose of illustrating fixed-point induction in PVS we choose the derivation of Hoare's *while* rule from the denotational semantics in Definition 4.3 using fixed-point induction.

Definition 4.4 (Hoare Triple). Let $D : \text{TYPE}^+$; then:

$$
\begin{array}{l}
p,\ q : \text{VAR } pred[D] \\
f \quad : \text{VAR } srel[D]
\end{array}
$$

$$\subseteq : preCPO[pred[D]] = \downarrow_{\leq}(predicates)$$

$$\models (p,\ f,\ q) : bool = (image(f)(p) \subseteq q)$$

Hoare-triples $\models (p,\ f,\ q)$ hold if the image of the function f with respect to precondition p is included, with respect to the ordering $\subseteq$ on predicates, in the postcondition q. Now, we have collected all ingredients to formulate and prove Hoare's *while* rule.

Lemma 4.5. Given the variable declarations from Definition 4.4, the following holds:

$$
\begin{array}{l}
\mathit{while\_rule} : \textsc{Lemma} \\
\quad \models (p \wedge b,\ f,\ p) \\
\Rightarrow \\
\quad \models (p,\ \mathit{while}(b,\ f),\ p \wedge \neg\, b)
\end{array}
$$

Unfolding the definition of *while* and propositional reasoning yields the following subgoal:[7]

$$
\begin{array}{l}
\mathit{while\_rule}\ : \\
\\
[-1] \quad \models (p!1 \wedge b!1,\ f!1,\ p!1) \\
\mid - - - - - - - \\
\{1\} \quad \models (p!1,\ \mu(\Psi(b!1,\ f!1)),\ p!1 \wedge \neg\, b!1)
\end{array}
$$

while_rule is proved using fixed-point induction *fp_induction_mono* from Theorem 3.15 by instantiating the latter's formal parameter P with

$$\lambda(F :\ \mathit{srel}[D]) : \models (p!1,\ F,\ p!1 \wedge \neg\, b!1)$$

Application of fixed-point induction yields three subgoals.

$$
\begin{array}{l}
\mathit{while\_rule}.1 : \\
\{-1\} \quad \models (p!1 \wedge b!1,\ f!1,\ p!1) \\
\mid - - - - - - - \\
\{1\} \quad \models (p!1, \mathit{abort}, p!1 \wedge \neg\, b!1)
\end{array}
$$

$$
\begin{array}{l}
\mathit{while\_rule}.2 : \\
\{-1\} \quad \models (p!1, x!1, p!1 \wedge \neg\, b!1) \\
[-2] \quad \models (p!1 \wedge b!1,\ f!1,\ p!1) \\
\mid - - - - - - - \\
\{1\} \quad \models (p!1, \\
\qquad \textsc{If}\ b!1\ \textsc{Then}\ f!1; x!1\ \textsc{Else}\ \mathit{skip}\ \textsc{Endif}\ , \\
\qquad p!1 \wedge \neg\, b!1)
\end{array}
$$

$$
\begin{array}{l}
\mathit{while\_rule}.3\ (\mathit{TCC}) : \\
\\
\mid - - - - - - - \\
\{1\} \quad \mathit{admissible}?(\lambda(F :\ \mathit{srel}[D]) : \models (p!1,\ F,\ p!1 \wedge \neg\, b!1))
\end{array}
$$

[7] Remember that proofs in Pvs are presented in a sequent calculus where antecedents and succedents arenumbered by negative and positive numbers, respectively.

Notice that the codomain of Ψ in 4.3 causes the prover to suppress a subgoal corresponding to the monotonicity of the functional $\Psi(b!1,\ f!1)$. Subgoals *while_rule*.1 and *while_rule*.2 respectively correspond to the induction base and induction step of the fixed-point induction rule; these subgoals are proved with less than 15 simple interactions such as unfolding of definitions and propositional reasoning. Furthermore, since the conclusion $P(\mu(srel[D])(f))$ of the fixed-point induction rule in Theorem 3.15 is constrained to admissible predicates P, an additional subgoal, a type correctness condition (TCC), *while_rule*.3 is generated. The critical idea in this admissibility proof is to characterize the least upper bound of chains C as follows:[8]

$$lub(\sqsubseteq)(C) = \lambda(s : D) : \bigcup(set\_image(C)(s))$$

This is possible, since the chain C 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

We have discussed various techniques for formalizing concepts of domain theory, including cpos, various domain constructions, monotonic functions, the fixed-point theorem for monotonic functions on cpos, and fixed-point induction. Although our encodings of fixed-point induction form a conservative extension of the underlying PVS logic, and consequently do not strengthen this logic, they permit natural formalization of many proofs by mixing fixed-point induction with inductions already built into PVS, like structural induction and well-founded induction.

Simple applications of these encodings indicate that it is rather straightforward to work with this theory. Other uses of formalized domain theory we have worked out include a comparison between various forms of semantics [15] and the verification of generic compilation schemes [6]. Altogether, these developments show that many interesting mathematical facts in the context of programming semantics can readily be formalized using current specification and theorem prover technology. This statement is underlined by the fact that a first version of these encodings was completed by the first author—who learned domain theory while formalizing it—within 2 months.

[8] $\bigcup(P : set[pred[D]]) :\ pred[D]\ =\ \lambda(d :\ D) :\ \exists(p : (P)) :\ p(d)$
$set\_image(F : set[[D \to R]])(x : D) :\ set[R]\ = \{y : R \mid \exists(f : (F)) :\ f(x) = y\}$

The main characteristics of our encodings is the systematic use of predicate subtypes and judgements; this drastically simplifies proofs, since many applicability conditions are deduced—during type-checking—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 simplify and streamline our proofs. Moreover, the lack of Hindley-Milner style polymorphism forces unnatural decomposition of theories.

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.

References

1. S. Agerholm. A HOL Basis for Reasoning about Functional Programs. Brics report series, Department of Computer Science, University of Aarhus, Denmark, 1994.
2. S. Agerholm. LCF Examples in HOL. *The Computer Journal*, 38(1), 1995.
3. F. Bartels, A. Dold, F.W. von Henke, H. Pfeifer, and H. Rueß. Formalizing Fixed-Point Theory in PVS. Ulmer Informatik-Berichte 96-10, Universität Ulm, December 1996.
4. R. Berghammer. Theoretische Grundlagen von Programmiersprachen und Programmentwicklung. Lecture Notes, 1996.
5. J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A Tutorial Introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995.
6. 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.
7. 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.
8. M.J.C Gordon and T.F. Melham. *Introduction to HOL : A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.
9. C.A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing Series. The MIT Press, 1992.
10. B. Knaster. Un Théorème sur les fonctions d'ensembles. *Annales de la Societé Polonaise de Mathématique*, 6:133–134, 1928.
11. J. Loeckx and K. Sieber. *The Foundations of Program Verification*. Series in Computer Science. Wiley-Teubner, second edition, 1987.

12. 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.
13. L.C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF.* Number 2 in Cambride Tracts in Theoretical Computer Science. Cambridge University Press, 1987.
14. L.C. Paulson. *Isabelle: A Generic Theorem Prover.* Number 828 in Lecture Notes in Computer Science. Springer-Verlag, 1994.
15. 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.
16. F. Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF.* PhD thesis, Technische Universität München, 1994.
17. 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.
18. D. A. Schmidt. *Denotational Semantics.* Wm. C. Brown Publishers, Dubuque, Iowa, 1988.
19. A. Tarski. A Lattice-Theoretic Fixpoint Theorem and its Applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
20. G. Winskel. *The Formal Semantics of Programming Languages.* Foundations of Computing Series. MIT Press, Cambridge, Massachusetts, 1993.

Address for correspondence: Harald Rueß, Universität Ulm, Fakultät für Informatik, James-Franck-Ring, 89069 Ulm, Germany. E-mail: `ruess@informatik.uni-ulm.de`.