Reasoning with Powerdomains in Isabelle/HOLCF

Brian Huffman

Portland State University
brianh@cs.pdx.edu

Abstract. This paper presents the first fully-mechanized formalization of powerdomains, implemented in the HOLCF logic of the Isabelle theorem prover. The powerdomain library provides an abstract view of powerdomains to the user, hiding the complicated implementation details. The library also provides proof automation, in the form of sets of rewrite rules for solving equalities and inequalities on powerdomains.

1 Introduction

Powerdomains are a domain-theoretic analog of powersets, which were designed for reasoning about the semantics of nondeterministic programs.[12] The use of powerdomains for reasoning about nondeterminism (and domain-theoretic denotational semantics in general) has declined in recent years, which I believe is primarily due to their perceived complexity. Compared to other more syntactic approaches to semantics, domain theory and powerdomains require a lot of mathematical sophistication to understand. This is a significant barrier for anyone who might want to use denotational semantics to reason about computation. It is my hope that the existence of good formalized libraries will remove that barrier to the use of domain theory for denotational semantics.

In this paper I attempt to demonstrate that powerdomains are a natural way for functional programmers to reason about nondeterministic programs. Using Haskell-style monadic code as a starting point, Section 2 motivates the definition of a powerdomain. Section 3 examines the three main varieties of powerdomains, and attempts to convey some intuitions about their structures and what each is good for. For readers wishing to use the powerdomain library, Section 4 documents all of the powerdomain operations provided by the library, as well as some of the lemmas and proof automation that is available. Section 5 describes the implementation of the powerdomain library; understanding this section is not necessary in order to use the library, and may be skipped on first reading.

This paper assumes some familiarity with the Haskell language. In particular, I expect the reader to know about monads, and the monad laws. I also assume that the reader is familiar with some of the basics of domain theory, which is traditionally used for reasoning denotationally about Haskell programs.[4] In particular, the reader should know about bottoms ($\bot$), complete partial orders ($\sqsubseteq$), limits of chains, continuous functions, and admissible predicates.

2 Nondeterminism monads

From a functional programmer's perspective, a powerdomain can be thought of as simply a special kind of monad for nondeterminism. In addition to the standard monad operations *return* and *bind*, a powerdomain also provides a binary operation for making a nondeterministic choice. In Haskell syntax, we can specify a subclass of monads that have such a binary choice operator:[10]

```
class (Monad m) => MultiMonad m where
  (+|+) :: m a -> m a -> m a
```

Haskell programmers often use the list monad to model nondeterministic computations; functions indicate multiple possible return values by enumerating them in a list. In this case, the list append operator $(+\!\!+)$ fills the role of nondeterministic choice.

```
instance MultiMonad [] where
  xs +|+ ys = xs ++ ys
```

The list monad has the great advantage of being executable: If you code up a nondeterministic algorithm in the list monad, you can just run it and see the results. However, for *reasoning* about nondeterministic algorithms, the list monad falls short in two important ways.

First, the list monad is not abstract enough: There are many different lists that represent the same set of possible return values. For example, consider a nondeterministic integer computation f with three possible outcomes: a return value of 3, a return value of 5, or divergence (i.e. a return value of $\bot$). The lists $[3, 5, \bot]$ and $[5, 5, 3, \bot, 3]$ both represent the value of f equally well; both represent the set $\{3, 5, \bot\}$. If divergence were not a possibility, then we could canonicalize the lists by sorting and removing duplicates—but obviously this does not work in general.

The second problem is that the list monad does not behave well in the presence of infinite or partial output. The problem originates with the definition of append: If xs is an infinite list, then $xs +\!\!+ ys$ does not depend on ys at all. If ys includes some possible outcomes that do not already occur in xs, then they get thrown away. Similarly, if xs is a partial list, like $1 : 2 : \bot$, then $xs +\!\!+ ys$ also ignores its second argument.

This problem is demonstrated by the following recursive nondeterministic computation. Any integer greater than or equal to 2 should be a possible result. However, when interpreted in the list monad, only even integers are included. The problem is that since the denotation of f is an infinite list, the "return 1" is never reached.

```
f :: (MultiMonad m) => m Int
f = do x <- return 0 +|+ f +|+ return 1
       return (x+2)
```

Another possible nondeterminism monad for Haskell is the binary tree, whose definition is shown below. The binary tree monad solves the second problem that lists had: Unlike the list append operator, the *Node* constructor never ignores either of its arguments, even if the other is partial or infinite. However, the problem of multiple representations remains; in fact this problem is even worse than before. Since the choice operator for trees is a data constructor, it doesn't satisfy any non-trivial equalities, while list append is at least associative.

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

instance Monad Tree where
  return x = Leaf x
  Leaf x >>= f = f x
  Node l r >>= f = Node (l >>= f) (r >>= f)

instance MultiMonad Tree where
  l +|+ r = Node l r
```

For doing formal reasoning about nondeterministic computations, an ideal nondeterminism monad should satisfy all the axioms listed in Fig. 1. We will call a monad that satisfies all seven laws a *powerdomain*. Laws 1–3 are just the standard Haskell monad laws. Law 4 says that bind distributes over the choice operator, and laws 5–7 state that choice is associative, commutative and idempotent. The list monad satisfies laws 1–5, but not 6 or 7; the binary tree monad satisfies only 1–4. There is no obvious way to define a true powerdomain directly in Haskell, but in the next section we will see how to define powerdomains mathematically.

1. `return x ≫= f = f x`
2. `xs ≫= return = xs`
3. `(xs ≫= f) ≫= g = xs ≫= (\x -> f x ≫= g)`
4. `(xs +|+ ys) ≫= f = (xs ≫= f) +|+ (ys ≫= f)`
5. `(xs +|+ ys) +|+ zs = xs +|+ (ys +|+ zs)`
6. `xs +|+ ys = ys +|+ xs`
7. `xs +|+ xs = xs`

Fig. 1. The powerdomain laws

Note that in addition to the seven powerdomain laws, there is another implicit requirement for powerdomains: All of the operations must be monotone and continuous, i.e. they must respect the cpo structure of the types on which they operate. In Haskell, every definable function is automatically continuous by construction, while in Isabelle, the logic permits the definition of functions which are not necessarily continuous. Continuity is a concept defined in Isabelle/HOLCF, and it is necessary to prove that each function defined in the library is continuous.

3 Powerdomains

There are multiple ways to define a powerdomain with operations that satisfy all of the desired laws. The three most common are known as the upper, lower, and convex powerdomains. These are also respectively known as the Smyth, Hoare, and Plotkin powerdomains. Historically, each variety is also associated with a musical symbol: sharp ($\sharp$) for upper, flat ($\flat$) for lower, and natural ($\natural$) for the convex powerdomain.

Before we dive into the details of the various powerdomains, first let us introduce some more notation. We will borrow the variable naming convention often used for lists in Haskell: For values of powerdomain types we use names like xs, ys, or zs, while for the underlying elements we use names like x, y, or z.

Also, we will consistently use set-style notation when talking about powerdomains. The singleton set syntax $\{-\}$ denotes the monadic return operator, "unit"; and the set union symbol $(\cup)$ denotes the nondeterministic choice operator, "plus". Also, we will use set enumerations like $\{x, y, z\}$ as shorthand for $\{x\} \cup \{y\} \cup \{z\}$. When necessary, we will indicate a specific powerdomain by using the appropriate musical symbol as a superscript.

3.1 Convex powerdomain

For a given element domain α, the convex powerdomain $\mathcal{P}^\natural(\alpha)$ is the *free continuous domain-algebra* over the constructors $\{-\}^\natural$ and $(\cup^\natural)$, modulo the associativity, commutativity, and idempotence of $(\cup^\natural)$. (This construction is explained in [1, §6.1]) The convex powerdomain is "universal" in a category-theoretical sense, in that there is a unique mapping (preserving unit and plus) from the convex powerdomain into any other powerdomain.

Freeness means two things here. First, it says that the convex powerdomain consists only of values that can be built up from applications of unit and plus (i.e. the convex powerdomain has "no junk"). Secondly, freeness also means that no nontrivial equalities between terms should hold, except those required by the laws (i.e. the convex powerdomain has "no confusion").

In the context of complete partial orders, the "no junk" property has a slightly different meaning than it does for ordinary inductive datatypes. As a cpo, the convex powerdomain includes values built from a finite number of constructor applications, plus additional values that result as limits of chains. Thus the convex powerdomain has an induction rule like the following:

$$\frac{\mathrm{adm}(P) \quad \forall x.\, P(\{x\}^\natural) \quad \forall xs\, ys.\, P(xs) \longrightarrow P(ys) \longrightarrow P(xs \cup^\natural ys)}{\forall xs.\, P(xs)} \tag{1}$$

Admissibility of P means that for any chain of elements x_i such that $P(x_i)$ holds for all i, P must also hold for the limit $\bigsqcup_i x_i$. This side condition reflects the fact that some values are only expressible as limits of chains—most induction rules in HOLCF have a similar admissibility side condition. (HOLCF can automatically prove admissibility for most inductive predicates used in practice.)

We still need to check that we can satisfy all of the powerdomain laws from Fig. 1. Laws 5–7 hold by construction. We can use laws 1 and 4 as defining equations for the bind operator. Finally, it is straightforward to prove laws 2 and 3 by induction.

Definition 1. *We say that x is a* member *of xs if $\{x\} \cup xs = xs$.*

If xs represents a nondeterministic computation, and x is one of the possible results, then x must be a member of xs. However, the set of members is not necessarily equal to the set of possible results. Not every conceivable set of results can be precisely represented in the convex powerdomain, as the following theorem implies.

Theorem 1. *Let xs be a value in a convex powerdomain. Then the set of members of xs is convex-closed.*

Proof. Let x and z be members of xs, and let y be any value between x and z, such that $x \sqsubseteq y$ and $y \sqsubseteq z$. We will show that y is a member of xs.

1. From $y \sqsubseteq z$, we have $\{y\}^\natural \cup^\natural xs \sqsubseteq \{z\}^\natural \cup^\natural xs$, by monotonicity.
   Then since z is a member of xs, we have $\{z\}^\natural \cup^\natural xs = xs$.
   Therefore $\{y\}^\natural \cup^\natural xs \sqsubseteq xs$.
2. From $x \sqsubseteq y$, we have $\{x\}^\natural \cup^\natural xs \sqsubseteq \{y\}^\natural \cup^\natural xs$, by monotonicity.
   Then since x is a member of xs, we have $\{x\}^\natural \cup^\natural xs = xs$.
   Therefore $xs \sqsubseteq \{y\}^\natural \cup^\natural xs$.

By antisymmetry we have $\{y\}^\natural \cup^\natural xs = xs$, thus y is a member of xs. □

Theorem 1 says that the set of members of xs includes at least the convex closure of the set of possible return values. In practice, this means that sometimes nondeterministic computations with different sets of possible outcomes nevertheless have the same denotation in the convex powerdomain.

Consider the domain of lifted booleans, which contains three values: True, False, and $\bot$. On top of this, we can construct the domain of pairs of booleans, which is ordered component-wise. Now imagine we have a nondeterministic computation f which has exactly two possible return values: either $(\text{True}, \text{False})$ or $(\bot, \bot)$. Next, define a computation g which additionally has a third possible return value of $(\text{True}, \bot)$. Here is how we might specify f and g in Haskell:

```
f, g :: (MultiMonad m) => m (Bool, Bool)
f = return (True, False) +|+ return (undefined, undefined)
g = return (True, undefined) +|+ f
```

If we model these computations using the convex powerdomain monad, then the denotation of f is $\{(\text{True, False}), (\bot, \bot)\}^\natural$, and the denotation of g is $\{(\text{True}, \bot), (\text{True, False}), (\bot, \bot)\}^\natural$. But according to Theorem 1, these values are actually equal—the convex powerdomain does not distinguish between the computations f and g. In general, two computations will be identified if their respective sets of possible results have the same convex closure.

This convex closure thing may seem a little weird; why bother with all this, when we could just represent multiple result values using ordinary sets? The weirdness is a small price to pay for a significant bonus: Since powerdomains are cpos, and all the operations are continuous, that means that we can freely use powerdomains with general recursion—something you cannot do with ordinary powersets.

3.2 Upper powerdomain

The upper powerdomain $\mathcal{P}^\sharp(\alpha)$ can be defined in the same manner as the convex powerdomain, except we require $(\cup^\sharp)$ to satisfy one extra law:

$$xs \cup^\sharp ys \sqsubseteq xs \tag{2}$$

(Note that due to commutativity, the statement $xs \cup^\sharp ys \sqsubseteq ys$ is equivalent.) This law makes the upper powerdomain into a semilattice, where $xs \cup^\sharp ys$ is the meet, or greatest lower bound, of xs and ys.

Theorem 2. *Let xs be a value in an upper powerdomain. Then the set of members of xs is upward-closed.*

Proof. Let x be a member of xs, and let y be any value such that $x \sqsubseteq y$. We will show that y is a member of xs.

1. From the symmetric form of Eq. 2, we have $\{y\}^\sharp \cup^\sharp xs \sqsubseteq xs$.
2. From $x \sqsubseteq y$, we have $\{x\}^\sharp \cup^\sharp xs \sqsubseteq \{y\}^\sharp \cup^\sharp xs$, by monotonicity.
   Then since x is a member of xs, we have $\{x\}^\sharp \cup^\sharp xs = xs$.
   Therefore $xs \sqsubseteq \{y\}^\sharp \cup^\sharp xs$.

By antisymmetry we have $\{y\}^\sharp \cup^\sharp xs = xs$, thus y is a member of xs. □

A consequence of this theorem is that if $\bot$ is a member of xs, then everything is a member of xs. In other words, if a nondeterministic computation has any possibility of returning $\bot$, then according to the upper powerdomain semantics, nothing else matters—it might as well *always* return $\bot$. For this reason, the upper powerdomain is good for reasoning about total correctness: if $\bot$ is *not* a member of xs, then you can be sure that xs denotes a computation that has no possibility of nontermination.

3.3 Lower powerdomain

The lower powerdomain $\mathcal{P}^\flat(\alpha)$ can also be defined similarly, by adding a different extra law:

$$xs \sqsubseteq xs \cup^\flat ys \tag{3}$$

This law makes the upper powerdomain into a semilattice, where $xs \cup^\flat ys$ is the join, or least upper bound, of xs and ys.

Theorem 3. *Let xs be a value in a lower powerdomain. Then the set of members of xs is downward-closed.*

Proof. Similar to the proof of Theorem 2. □

An immediate consequence of this theorem is that in the lower powerdomain, $\perp$ is a member of everything. Equivalently, $\{\perp\}^\flat$ is an identity for the $(\cup^\flat)$ operation. In terms of nondeterministic computations, this means that the lower powerdomain semantics ignores any nonterminating execution paths. In contrast to the upper powerdomain, the lower powerdomain is better for reasoning about partial correctness, where you want to verify that *if* a computation terminates, then its result will satisfy some property.

3.4 Visualizing powerdomains

To help convey an intuition for the structure of the various kinds of powerdomains, this section includes diagrams of the powerdomain orderings over a few different element types. Fig. 2 shows all three powerdomains over a small flat domain, like the lifted booleans. Fig. 3 extends this to a slightly larger flat domain. Fig. 4 extends this in a different way by adding a top value.

Looking at Figs. 2 and 3, some generalizations can be made about powerdomains over flat cpos. The ordering on the lower powerdomain of any flat cpo is isomorphic to the subset ordering on the corresponding powerset. Also note that the lower powerdomain always has a greatest element, which corresponds to the set including all possible return values. In contrast, the upper powerdomain is almost like the lower powerdomain flipped upside-down, except that the bottom element stays at the bottom; the other singleton sets are maximal in this ordering.

For the lifted two-element type, note that the convex powerdomain has the structure of the lower powerdomain embedded inside it, but with a new value (excluding $\perp$) added above each old value. The convex powerdomain of the lifted three-element type is not shown (due to its size) but it is related to the lower powerdomain in the same way.

The four-element lattice is interesting because due to its symmetry, it clearly illustrates the duality between the upper and lower powerdomains. The lower powerdomain is structured exactly like the upper powerdomain, but with the order reversed.

4 HOLCF powerdomain library

This section describes the user-visible aspects of the HOLCF powerdomain library. The implementation defines three new type constructors, one for each of the three powerdomain varieties. Each type has `unit` and `plus` constructors, and a monadic `bind` operator. Each type also has `map` and `join` operators, defined in terms of `unit` and `bind` in the same manner as Haskell's `liftM` and `join`. The full list of types and constants is shown in Fig. 5.

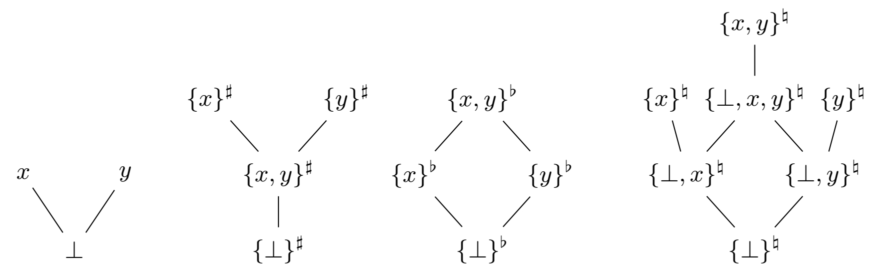

Fig. 2. Lifted two-element type, with upper, lower, and convex powerdomains

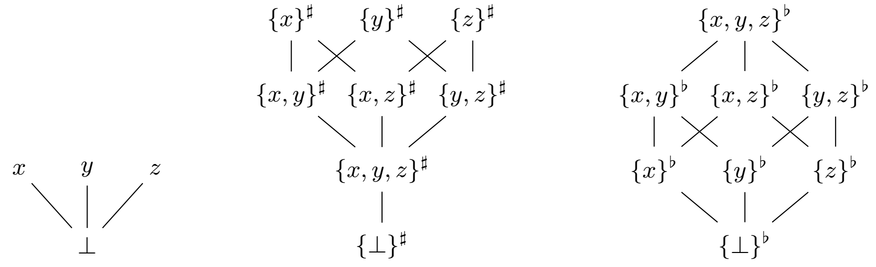

Fig. 3. Lifted three-element type, with upper and lower powerdomains

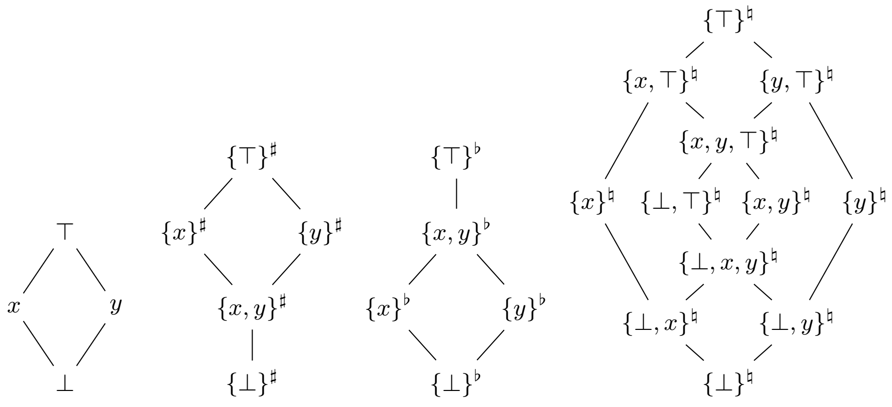

Fig. 4. Four-element lattice, with upper, lower, and convex powerdomains

The functions convex_to_lower and convex_to_upper are the mappings guaranteed to exist by the universal property of the convex powerdomain; they preserve unit and plus. Note that instead of the full function space (=>), all functions use the HOLCF continuous function space type (->), indicating that they are continuous functions.

```
typedef 'a upper_pd
upper_unit :: 'a -> 'a upper_pd
upper_plus :: 'a upper_pd -> 'a upper_pd -> 'a upper_pd
upper_bind :: 'a upper_pd -> ('a -> 'b upper_pd) -> 'b upper_pd
upper_map :: ('a -> 'b) -> 'a upper_pd -> 'b upper_pd
upper_join :: 'a upper_pd upper_pd -> 'a upper_pd

typedef 'a lower_pd
lower_unit :: 'a -> 'a lower_pd
lower_plus :: 'a lower_pd -> 'a lower_pd -> 'a lower_pd
lower_bind :: 'a lower_pd -> ('a -> 'b lower_pd) -> 'b lower_pd
lower_map :: ('a -> 'b) -> 'a lower_pd -> 'b lower_pd
lower_join :: 'a lower_pd lower_pd -> 'a lower_pd

typedef 'a convex_pd
convex_unit :: 'a -> 'a convex_pd
convex_plus :: 'a convex_pd -> 'a convex_pd -> 'a convex_pd
convex_bind :: 'a convex_pd -> ('a -> 'b convex_pd) -> 'b convex_pd
convex_map :: ('a -> 'b) -> 'a convex_pd -> 'b convex_pd
convex_join :: 'a convex_pd convex_pd -> 'a convex_pd
convex_to_upper :: 'a convex_pd -> 'a upper_pd
convex_to_lower :: 'a convex_pd -> 'a lower_pd
```

Fig. 5. Powerdomain types and constants defined in HOLCF

For convenience, the library also provides set-style syntax for unit and plus, similar to the notation used in this paper.

Along with the definitions of types and constants, the library provides a significant body of lemmas. Each powerdomain type has an induction rule in terms of unit and plus. Rules about injectivity, strictness, compactness, and ordering are provided for the constructors. Also, the functor and monad laws are provided as lemmas.

4.1 Bifinite type class

HOLCF uses Isabelle's axiomatic type class mechanism [16] to represent different kinds of domains. The main axiomatic type classes in HOLCF are cpo (chain-complete partial orders) and pcpo (pointed cpos). Unfortunately, the powerdomain constructions do not work over arbitrary cpos; they need some additional structure. In order to formalize powerdomains in HOLCF, it was necessary to add a new axiomatic class `bifinite`, which is a subclass of `pcpo`. I will have more to say about class `bifinite` in Section 5.

As far as a user of the library is concerned, it does not matter how class `bifinite` is defined; the important thing is that it should be preserved by all of type constructors that the user works with. In the current version of Isabelle, instances are provided for all type constructors defined in the HOLCF library: continuous function space, Cartesian product, strict product, strict sum, lifted cpos, and all three varieties of powerdomains. Flat domains built from countable HOL types are instances of `bifinite` as well.

A known problem is that the current implementation of the domain package does not generate instances of class `bifinite` for new types. In the current version of HOLCF, if a user wants to use a domain package–defined type with powerdomains, it will be necessary to manually prove that the type is an instance of class `bifinite`. Updating the domain package to work with the `bifinite` class is planned as future work.

4.2 Automation

To facilitate reasoning with powerdomains, the library provides various sets of rewrite rules that are designed to work well together.

ACI normalization. Isabelle's simplifier is set up to handle permutative rewrite rules. For any associative-commutative operator, there is a set of three permutative rewrite rules that can convert any expression built from the operator into a normal form (grouped to the right, with terms sorted according to some term-ordering).[2] Two of the AC rewrites are simply the associativity and commutativity rules. The third is the left-commutativity rule. For ACI rewriting, we need a total of five rules: the three AC rewrites, plus the idempotency rule, and also (analogous to left-commutativity) left-idempotency.

$$\begin{aligned} (xs \cup ys) \cup zs &= xs \cup (ys \cup zs) \\ ys \cup xs &= xs \cup ys \\ ys \cup (xs \cup zs) &= xs \cup (ys \cup zs) \\ xs \cup xs &= xs \\ xs \cup (xs \cup ys) &= xs \cup ys \end{aligned} \tag{4}$$

Permutative rewriting using the ACI rules results in a normal form where expressions are nested to the right, and the terms are in sorted order, with no exact duplicates. In HOLCF, this normalization can be accomplished for the convex powerdomains by invoking `(simp add: convex_plus_aci)`. Similarly, `upper_plus_aci` and `lower_plus_aci` may be used with upper and lower powerdomains, respectively.

Solving inequalities. A common subgoal in a proof might be to show that one powerdomain expression approximates another. For each variety of powerdomain, there is a set of rewrites that can be used to automatically reduce an inequality on powerdomains down to inequalities on the underlying type.

$$\begin{aligned} \{x\}^\sharp \sqsubseteq \{y\}^\sharp &\iff x \sqsubseteq y \\ xs \sqsubseteq (ys \cup^\sharp zs) &\iff (xs \sqsubseteq ys) \wedge (xs \sqsubseteq zs) \\ (xs \cup^\sharp ys) \sqsubseteq \{z\}^\sharp &\iff (xs \sqsubseteq \{z\}^\sharp) \vee (ys \sqsubseteq \{z\}^\sharp) \end{aligned} \tag{5}$$

$$\begin{aligned} \{x\}^\flat \sqsubseteq \{y\}^\flat &\iff x \sqsubseteq y \\ (xs \cup^\flat ys) \sqsubseteq zs &\iff (xs \sqsubseteq zs) \wedge (ys \sqsubseteq zs) \\ \{x\}^\flat \sqsubseteq (ys \cup^\flat zs) &\iff (\{x\}^\flat \sqsubseteq ys) \vee (\{x\}^\flat \sqsubseteq zs) \end{aligned} \tag{6}$$

$$\begin{aligned} \{x\}^\natural \sqsubseteq \{y\}^\natural &\iff x \sqsubseteq y \\ \{x\}^\natural \sqsubseteq (ys \cup^\natural zs) &\iff (\{x\}^\natural \sqsubseteq ys) \wedge (\{x\}^\natural \sqsubseteq zs) \\ (xs \cup^\natural ys) \sqsubseteq \{z\}^\natural &\iff (xs \sqsubseteq \{z\}^\natural) \wedge (ys \sqsubseteq \{z\}^\natural) \end{aligned} \tag{7}$$

For the upper and lower powerdomains, each has a set of three rewrite rules that covers all cases of comparisons. For example, invoking `(simp add: upper_pd_less_simps)` will rewrite $\{x, y\}^\sharp \sqsubseteq \{y, z\}^\sharp$ to $x \sqsubseteq z \vee y \sqsubseteq z$, using the rules in Eq. (5). Similarly, `(simp add: lower_pd_less_simps)` uses the rules in Eq. (6) to simplify inequalities on lower powerdomains.

For the convex powerdomain, the three rules in Eq. (7) are incomplete: They do not cover the case of $(xs \cup^\natural ys) \sqsubseteq (zs \cup^\natural ws)$. To handle this case, we will take advantage of the coercions from the convex powerdomain to the upper and lower powerdomains, along with the following ordering property:

$$xs \sqsubseteq ys \iff \text{to\_upper}(xs) \sqsubseteq \text{to\_upper}(ys) \wedge \text{to\_lower}(xs) \sqsubseteq \text{to\_lower}(ys) \tag{8}$$

The rule set `convex_pd_less_simps` includes all rules from Eqs. (5)–(7), and a suitably instantiated Eq. (8) to cover the missing case.

Using inequalities to solve non-trivial equalities. The ACI rewriting can take care of many equalities between powerdomain expressions, but the inequality rules can actually solve more. For example, using the assumptions $x \sqsubseteq y$ and $y \sqsubseteq z$, we will prove that $\{x, y, z\}^\natural = \{x, z\}^\natural$. By antisymmetry, we can rewrite this to the conjunction $(\{x, y, z\}^\natural \sqsubseteq \{x, z\}^\natural) \wedge (\{x, z\}^\natural \sqsubseteq \{x, y, z\}^\natural)$. Next, we can use the method `(simp add: convex_pd_less_simps)`, and this subgoal reduces to $(y \sqsubseteq x \vee y \sqsubseteq z) \wedge (x \sqsubseteq y \vee z \sqsubseteq y)$. Finally, this is easily discharged using the assumptions $x \sqsubseteq y$ and $y \sqsubseteq z$.

5 Implementation

The development of powerdomains in HOLCF follows the ideal completion construction presented by Gunter and Scott in [5, §5.2]. Some alternative constructions are also given by Abramsky and Jung in [1, §6.2]; the ideal completion method was chosen because it required the formalization of a minimal amount of supporting theories, and it offered good opportunities for proof reuse.

5.1 Class of bifinite domains

The powerdomain construction used in HOLCF makes use of an alternative representation of domains, where we just consider the set of compact (i.e. finite) values, rather than the whole domain.[1, §2.2.6] For this representation to work, we restrict our attention to *algebraic* cpos, where every value can be expressed as the limit of its compact approximants. This means that in an algebraic cpo the set of compact elements, together with the domain ordering on them, fully represents the entire domain. We say that the set of compact elements forms a *basis* for the domain, and the entire domain is a *completion* of the basis.

Most of HOLCF has been designed using the type class `pcpo` of pointed complete partial orders. However, `pcpo` types are not algebraic in general, and the ideal completion construction only works with algebraic cpos. Therefore it was necessary to add a new type class to HOLCF.

The class `bifinite` is defined as follows. It fixes a sequence of functions approx_n, and assumes four class axioms:

1. The approx_n form a chain
2. The least upper bound ($\bigsqcup_n \text{approx}_n$) is the identity function
3. Each approx_n is idempotent
4. Each approx_n has finite range

The HOLCF `bifinite` class actually corresponds to the "ω-bifinite" domains, which have a countable basis; the usual definition of "bifinite" [1, §4.2] has no such restriction, and would be equivalent to allowing any directed set of approx functions, rather than a countable chain. Bifinite domains were originally defined by Plotkin as limits of expanding sequences of finite posets, who used the name "SFP domain".[12]

Of all the various classes of domains to choose from, the definition of `bifinite` was chosen for the following reasons:

- All `bifinite` types are algebraic: Every `bifinite` type has a countable basis of compact elements, given by the union of the ranges of the approx functions.
- In `bifinite` types, every directed set contains a chain with the same limit. This means that in class `bifinite`, the notions of directed-continuity and chain-continuity coincide. This is important for fitting the ideal completion construction (which uses directed sets) into HOLCF (which defines everything with chains).
- The `bifinite` class is closed under all type constructors used in HOLCF, including the convex powerdomain.

5.2 Ideal completion

Given a basis $\langle B, \preceq \rangle$, we can reconstruct the full algebraic cpo. The standard process for doing this is called ideal completion, and it is done by considering the set of ideals over the basis:

Definition 2. *A set S is an* ideal *with respect to partial preorder relation* $(\preceq)$ *if it has the following properties:*

- *S is nonempty:* $\exists x.\, x \in S$
- *S is downward-closed:* $\forall x\, y.\, x \preceq y \longrightarrow y \in S \longrightarrow x \in S$
- *S is directed (i.e. has an upper bound for any pair of elements):* $\forall x\, y.\, x \in S \longrightarrow y \in S \longrightarrow (\exists z.\, z \in S \wedge x \preceq z \wedge y \preceq z)$

A principal ideal *is an ideal of the form* $\{y.\, y \preceq x\}$ *for some x, denoted* $\downarrow x$.

The set of all ideals over $\langle B, \preceq \rangle$ is denoted $\mathrm{Idl}(B)$; when ordered by subset inclusion, $\mathrm{Idl}(B)$ forms an algebraic cpo. The compact elements of $\mathrm{Idl}(B)$ are exactly those represented by principal ideals.

Note that the relation $(\preceq)$ does not need to be antisymmetric. For x and y that are equivalent (that is, both $x \preceq y$ and $y \preceq x$) the principal ideals $\downarrow x$ and $\downarrow y$ are equal. This means that the ideal completion construction automatically takes care of quotienting by the equivalence induced by $(\preceq)$.

The ideal completion construction is formalized in HOLCF using Isabelle's locale mechanism.[8] The library defines a locale `preorder` that fixes a type corresponding to the basis B, and a preorder relation on that type; within this locale, a predicate `ideal` is defined. Within the `preorder` locale, the main lemma proved is that the union of a chain of ideals is itself an ideal—which shows that the ideal completion is a cpo.

All three of the powerdomains in the library are defined by ideal completion. For an basis, the library defines a type `'a pd_basis`, which consists of nonempty, finite sets of compact elements of type `'a`. Following [5, §5.2], each of the three powerdomains is defined as an ideal completion over the same basis, but each uses a different preorder relation:

$$\begin{aligned} a \preceq^{\flat} b &\iff \forall x \in a.\ \exists y \in b.\ x \sqsubseteq y \\ a \preceq^{\sharp} b &\iff \forall y \in b.\ \exists x \in a.\ x \sqsubseteq y \\ a \preceq^{\natural} b &\iff a \preceq^{\flat} b \wedge a \preceq^{\sharp} b \end{aligned} \tag{9}$$

5.3 Continuous extensions of functions

A continuous function on an algebraic cpo is completely determined by its action on compact elements. This suggests a method for defining continuous functions over ideal completions: First, define a function from the basis B to a cpo C such that f is monotone, i.e. $x \preceq y$ implies $f(x) \sqsubseteq f(y)$. Then there exists a unique function $\hat{f} : \mathrm{Idl}(B) \to C$ that agrees with f on principal ideals, i.e. for all x, $\hat{f}(\downarrow x) = f(x)$. We say that $\widehat{f}$ is the *continuous extension* of f.

On top of the `preorder` locale, HOLCF defines another locale `ideal_completion` which fixes a second type corresponding to $\mathrm{Idl}(B)$. It also fixes a function `principal` of type $B \to \mathrm{Idl}(B)$. Within this locale, a function `basis_fun` is defined, which takes a monotone function f as an argument, and returns the continuous extension $\widehat{f}$.

The continuous extension is defined by mapping the function f over the input ideal, and then taking the least upper bound of the resulting directed set: $\widehat{f}(S) = \bigsqcup_{x \in S} f(x)$. Ordinarily, the result type C would need to be a directed-complete partial order to ensure that this least upper bound exists; however, the HOLCF library uses a different method which allows C to be any chain-complete partial order.

HOLCF defines a third locale `basis_take`, which fixes a chain of `take` functions over the basis elements—it is like a version of the `bifinite` class for bases. The `basis_take` locale ensures that the ideal completion $\mathrm{Idl}(B)$ is a bifinite domain. It is also used with the definition of `basis_fun` to construct a chain with the same limit as the directed set $\bigsqcup_{x \in S} f(x)$, which allows C to be an arbitrary chain-cpo.

The `basis_fun` combinator is used to define the powerdomain constructors `unit` and `plus` in terms of the singleton and union operations on the `pd_basis` type. The `bind` operators are also defined using `basis_fun`, in terms of a finite-set fold operation on `pd_basis`. Finally, to prove the `bifinite` class instance, the approx functions are also defined with `basis_fun`, in terms of the `take` functions on `pd_basis`.

5.4 Transferring properties to the completed domain

Once the powerdomain types are defined using ideal completion, with operations defined by continuous extension, the final step is to prove the relevant lemmas. For example, consider the lower powerdomain law $xs \sqsubseteq xs \cup^\flat ys$. In the case where xs and ys are both compact (i.e. represented by principal ideals) the proof follows easily from the definitions. Since $xs \sqsubseteq xs \cup^\flat ys$ is an admissible predicate on both xs and ys, this is in fact sufficient to show that it holds for all xs and ys.

Other properties are more tricky to transfer. For example, consider the rule $\{x\}^\natural \sqsubseteq \{y\}^\natural \Longrightarrow x \sqsubseteq y$. As before, this property is easy to prove for compact x and y. However, we cannot immediately infer that it holds for all x and y, since (because of the implication) this is not an admissible predicate.

The proof of $\{x\}^\natural \sqsubseteq \{y\}^\natural \Longrightarrow x \sqsubseteq y$ requires a few extra steps, making use of the approx functions from the `bifinite` class: To prove $x \sqsubseteq y$, it will be sufficient to show that for all n, $\mathrm{approx}_n x \sqsubseteq \mathrm{approx}_n y$. Now, from $\{x\}^\natural \sqsubseteq \{y\}^\natural$ we have $\mathrm{approx}_n \{x\}^\natural \sqsubseteq \mathrm{approx}_n \{y\}^\natural$, by monotonicity; then from the definition of approx on the convex powerdomain, this simplifies to $\{\mathrm{approx}_n x\}^\natural \sqsubseteq \{\mathrm{approx}_n y\}^\natural$. Finally, since $\mathrm{approx}_n x$ and $\mathrm{approx}_n y$ are compact, we can easily show that $\mathrm{approx}_n x \sqsubseteq \mathrm{approx}_n y$. All of the rules listed in Eqs. (5)–(8) use a similar proof.

6 Related work

There are several theorem prover formalizations of domain theory in existence. The current development is built on top of HOLCF, originally implemented by Regensburger, and later extended by many others.[13,9] HOLCF does not formalize very many different classes of domains; most concepts are defined in terms of pointed chain-complete partial orders and chain-continuity, which is the minimum amount of structure required to define a fixed-point combinator. It is intended to be used as a library for users to define datatypes and recursive functions and algorithms on them.

In the mid-1990s a group from the University of Ulm formalized parts of domain theory in PVS.[3] Its design goals appear to be similar to HOLCF—it includes just enough of domain theory to formalize fixed-points and fixed-point induction.

A formalization of domain theory with rather different goals is "Elements of Domain Theory", implemented in Coq in the 1990s by Kahn. It is based on the definitions and lemmas from [7]. This development defines several classes of domains, including directed-complete partial orders, omega-algebraic cpos, and bounded-complete domains. However, it does not define any type constructors. In contrast to HOLCF, it does not appear to be application-oriented; it seems the main intent was to formalize the textbook-style definitions and lemmas from the paper.

Other formalizations use a different logic to ensure that all functions are continuous by construction, such as the LCF system by Paulson.[11] Another interesting approach is taken by Reus with his development of synthetic domain theory in LEGO.[14] Instead of defining classes of domains in terms of a domain ordering, it starts by introducing a *subobject classifier*, which is characterized by a collection of axioms. The soundness of the construction is justified by a separate model.

Relevant uses of powerdomains include modeling interleaved and parallel computation. Papaspyrou uses the convex powerdomain, together with the state and resumption monad transformers, to model impure languages with unspecified evaluation order.[10] Along similar lines, Thiemann used a type of state monad built on top of powerdomains to reason about concurrent computations.[15] The monad transformers used in these works, specifically the resumption monad transformer, have been studied in HOLCF by Huffman, et al.[6]

7 Conclusion and future work

The powerdomain library described here is included as part of Isabelle2008 theorem prover. It can already be used to prove properties of simple nondeterministic algorithms, with automation for certain kinds of subgoals. Future work will focus on better integration with the HOLCF domain package: Bifinite class instances must be generated for all new datatypes. Also, the domain package needs to be extended to allow recursive type definitions involving powerdomains—this will enable the use of powerdomains for modeling parallel computation and concurrency.

References

1. Samson Abramsky and Achim Jung. Domain theory. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science Volume 3*, pages 1–168. Oxford University Press, 1994.
2. Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
3. F. Bartels, A. Dold, F. W. v. Henke, H. Pfeifer, and H. Rueß. Formalizing Fixed-Point Theory in PVS. Ulmer Informatik-Berichte 96-10, Universität Ulm, Fakultät für Informatik, 1996.
4. Richard Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall PTR, 2 edition, May 1998.
5. Carl A. Gunter and Dana S. Scott. Semantic domains. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B)*, pages 633–674. 1990.
6. Brian Huffman, John Matthews, and Peter White. Axiomatic constructor classes in Isabelle/HOLCF. In *In Proc. 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs '05), Volume 3603 of Lecture Notes in Computer Science*, pages 147–162. Springer, 2005.
7. Gilles Kahn, Inria Sophia Antipolis, Gordon D. Plotkin, and George Gratzer. Concrete domains. *Theoretical Computer Science*, 121:121–1, 1993.
8. Florian Kammuller, Markus Wenzel, and Lawrence C. Paulson. Locales: A sectioning concept for isabelle. In *Theorem Proving in Higher Order Logics (TPHOLs '99), LNCS 1690*, pages 149–165. Springer, 1999.
9. Olaf Müller, Tobias Nipkow, David von Oheimb, and Oskar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.
10. Nikolaos Papaspyrou and Dragan Macos. A study of evaluation order semantics in expressions with side effects. *Journal of Functional Programming*, 10(3):227–244, 2000.
11. Lawrence C. Paulson. *Logic and computation: interactive proof with Cambridge LCF.* Cambridge University Press, New York, NY, USA, 1987.
12. Gordon D. Plotkin. A powerdomain construction. *SIAM J. Comput.*, 5(3):452–487, 1976.
13. Franz Regensburger. HOLCF: Higher Order Logic of Computable Functions. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, Aspen Grove, Utah, 1995. Springer-Verlag LNCS 971.
14. Bernhard Reus. Synthetic domain theory in type theory: Another logic of computable functions. In *TPHOLs '96: Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, pages 365–380, London, UK, 1996. Springer-Verlag.
15. Peter Thiemann. Towards a denotational semantics for concurrent state transformers, 1995.
16. Markus Wenzel. Type classes and overloading in higher-order logic. In E. Gunter and A. Felty, editors, *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97)*, pages 307–322, Murray Hill, New Jersey, 1997.