Reasoning with Powerdomains in Isabelle/HOLCF

Brian Huffman

Department of Computer Science,
Portland State University, Portland, Oregon
brianh@cs.pdx.edu

Abstract

- *First fully-mechanized formalization of powerdomains*
- *Implemented in HOLCF logic of domain theory, in the Isabelle theorem prover*
- *Library hides complicated implementation details*
- *Proof automation for solving equalities and inequalities*

1. Introduction

- Powerdomains are a domain-theoretic analog of powersets, which were designed for reasoning about the semantics of nondeterministic programs.
- A powerdomain provides all of the operations of a monad. In addition, it provides a binary operation for making a nondeterministic choice.
- As part of domain theory, we can freely combine nondeterminism with higher-order functions and arbitrary recursion.

2. Lists for Nondeterminism

In Haskell, the **lazy list monad** is often used to model nondeterministic computations. The **append** function models nondeterministic choice.

```
incdec :: Int -> [Int]
incdec x = return (x+1) ++ return (x-1)
-- increment OR decrement the argument x
```

Lists are useful because they are **executable**. But they are not a good **denotational model for nondeterminism**, for several reasons:

- **Append is not commutative.** The following program produces the same set of results as `incdec`; ordering should not matter.

```
decinc :: Int -> [Int]
decinc x = return (x-1) ++ return (x+1)
```

- **Append is not idempotent.** The following two programs have the same set of possible results; repetition should not matter.

```
prog1, prog2 :: [Int]
prog1 = incdec 5 ++ incdec 3
prog2 = return 6 ++ return 4 ++ return 2
```

- **Append favors its left argument.** In case the left argument is partial or infinite, the right argument is ignored. In the recursive function `f` below, the `(x-1)` branch is never reached. In fact, `f` and `g` are equivalent!

```
f, g :: Int -> [Int]
f x = return x ++ f (x+1) ++ f (x-1)
g x = return x ++ g (x+1)
```

The definition of powerdomain in the next section addresses all of these limitations of the list monad.

3. Axiomatization of Powerdomains

A **powerdomain** is a monad with a **nondeterministic choice** operator, which is **associative**, **commutative**, and **idempotent**.

Powerdomain Operations

- Singleton (i.e. monadic unit/return) $\{-\}$
- Binary choice operator $- \uplus -$
- Monadic bind operator $- \gg\!= -$

All operations must be **continuous**.

Powerdomain Laws

1. Left unit: $\{x\} \gg\!= f = f(x)$
2. Right unit: $xs \gg\!= (\lambda x.\ \{x\}) = xs$
3. Bind-assoc: $(xs \gg\!= f) \gg\!= g = xs \gg\!= (\lambda x.\ f(x) \gg\!= g)$
4. Bind-plus: $(xs \uplus ys) \gg\!= f = (xs \gg\!= f) \uplus (ys \gg\!= f)$
5. Plus-assoc: $(xs \uplus ys) \uplus zs = xs \uplus (ys \uplus zs)$
6. Plus-comm: $xs \uplus ys = ys \uplus xs$
7. Plus-idem: $xs \uplus xs = xs$

Laws 1–3 are the standard **monad laws** from Haskell. The lazy list monad with append satisfies only Laws 1–5.

4. Convex Powerdomain

Given an element domain D, we can define the convex powerdomain $P^\natural(D)$ as a **free domain-algebra**:

1. Define a **recursive datatype**, with $\{-\}^\natural$ and $- \uplus^\natural -$ as **constructors**.
2. **Quotient** this datatype modulo **associativity**, **commutativity**, and **idempotence** of $- \uplus^\natural -$ (Laws 5–7).
3. Use Laws 1 and 4 as **defining equations** for bind.
4. Prove Laws 2 and 3 by **induction** over xs.

Thus $P^\natural(D)$ satisfies all seven powerdomain laws.

- $P^\natural(D)$ is **universal** in a category-theoretical sense
- Unique powerdomain **homomorphism** from $P^\natural(D)$ to any other powerdomain of D
- $P^\natural(D)$ **distinguishes** as many values as possible
- $P^\natural(D)$ **identifies** computations whose sets of results have the same **convex-closure** (see Figure 1)

5. Upper Powerdomain

We can define the upper powerdomain $P^\sharp(D)$ as a free domain-algebra satisfying an **additional law**:

$$xs \uplus^\sharp ys \sqsubseteq xs$$

- $xs \uplus^\sharp ys$ is **greatest lower bound** of xs and ys
- $xs \sqsubseteq ys$ if ys has **fewer** possible outcomes than xs
- Binary choice is **strict**: $\bot \uplus^\sharp xs = \bot$
- **Demonic nondeterminism**: "Possibly not terminating is just as bad as never terminating"
- Good for reasoning about **total correctness**
- $P^\sharp(D)$ **identifies** computations whose sets of results have the same **upward-closure**

6. Lower Powerdomain

We can define the lower powerdomain $P^\flat(D)$ as a free domain-algebra satisfying an **additional law**:

$$xs \sqsubseteq xs \uplus^\flat ys$$

- $xs \uplus^\flat ys$ is **least upper bound** of xs and ys
- $xs \sqsubseteq ys$ if ys has **more** possible outcomes than xs
- $\bot$ is **identity** for binary choice: $\bot \uplus^\flat xs = xs$
- **Angelic nondeterminism**: "I don't care about execution paths that don't terminate"
- Good for reasoning about **partial correctness**
- $P^\flat(D)$ **identifies** computations whose sets of results have the same **downward-closure**

7. Visualizing Powerdomains

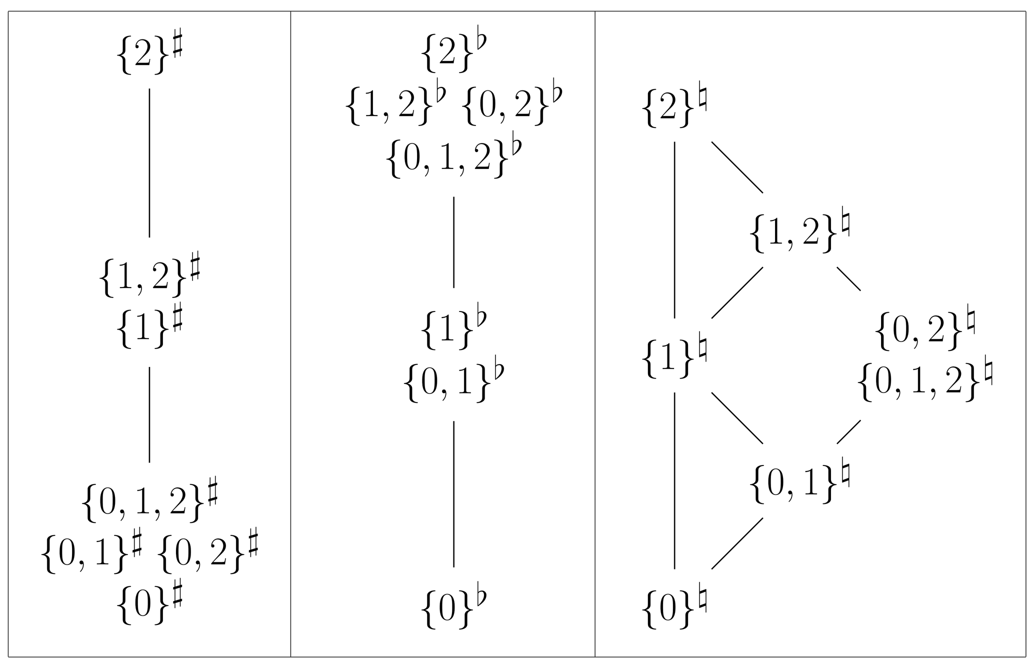

Figure 1: *Upper, lower, and convex powerdomains of the three-element linearly ordered domain* $0 \sqsubseteq 1 \sqsubseteq 2$

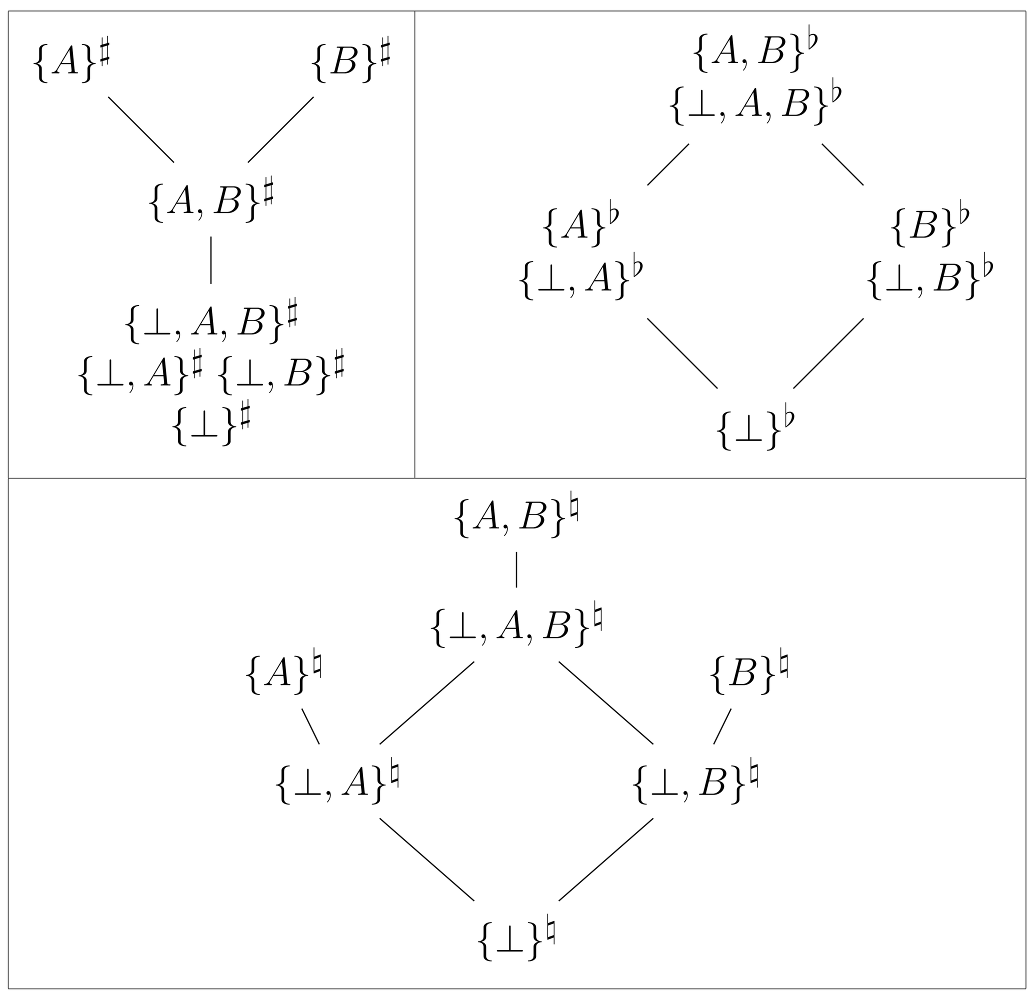

Figure 2: *Upper, lower, and convex powerdomains of the lifted two-element type* $A \quad \bot \quad B$

8. Proof Automation

ACI Rewriting

Isabelle can use **permutative rewrite rules** to sort elements and remove duplicates in powerdomain expressions.

$$\begin{aligned}(xs \uplus ys) \uplus zs &= xs \uplus (ys \uplus zs)\\ ys \uplus xs &= xs \uplus ys\\ ys \uplus (xs \uplus zs) &= xs \uplus (ys \uplus zs)\\ xs \uplus xs &= xs\\ xs \uplus (xs \uplus ys) &= xs \uplus ys\end{aligned}$$

- ACI rules can solve goals like $\{z, y, x, x, y\}^\sharp = \{x, y, z\}^\sharp$

Solving Inequalities

Rewrite rules can reduce inequalities on **powerdomains** to inequalities on the **element type**.

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

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

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

- The inequality rewrite rules can reduce subgoals like $\{x, y\}^\sharp \sqsubseteq \{y, z\}^\sharp$ to $x \sqsubseteq z \vee y \sqsubseteq z$
- With **antisymmetry**, these rules can solve equalities too

21st International Conference on Theorem Proving in Higher Order Logics, Montréal, Québec, Canada, 18–21 August 2008