1-1-2011

HOLCF '11: A Definitional Domain Theory for Verifying Functional Programs

Brian Charles Huffman
Portland State University

10.15760/etd.113

HOLCF '11: A Definitional Domain Theory for Verifying Functional Programs

by

Brian Charles Huffman

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Dissertation Committee:
James Hook, Chair
John Matthews
Mark Jones
Tim Sheard
Gerardo Lafferriere

Portland State University

ABSTRACT

HOLCF is an interactive theorem proving system that uses the mathematics of domain theory to reason about programs written in functional programming languages. This thesis introduces HOLCF '11, a thoroughly revised and extended version of HOLCF that advances the state of the art in program verification: HOLCF '11 can reason about many program definitions that are beyond the scope of other formal proof tools, while providing a high degree of proof automation. The soundness of the system is ensured by adhering to a definitional approach: New constants and types are defined in terms of previous concepts, without introducing new axioms.

Major features of HOLCF '11 include two high-level definition packages: the Fixrec package for defining recursive functions, and the Domain package for defining recursive datatypes. Each of these uses the domain-theoretic concept of least fixed points to translate user-supplied recursive specifications into safe low-level definitions. Together, these tools make it easy for users to translate a wide variety of functional programs into the formalism of HOLCF. Theorems generated by the tools also make it easy for users to reason about their programs, with a very high level of confidence in the soundness of the results.

As a case study, we present a fully mechanized verification of a model of concurrency based on powerdomains. The formalization depends on many features unique to HOLCF '11, and is the first verification of such a model in a formal proof tool.

ACKNOWLEDGMENTS

I would like to thank my advisor, John Matthews, for having continued to devote so much time to working with me, even as a part-time professor; and for motivating me to keep studying domain theory (and enjoying it!) these past years.

CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

This is a thesis about program verification—how to make sure that your computer programs do exactly what they are supposed to do. In other words, our goal is to be able to specify a computer program, state properties about it, and then construct proofs of those properties.

To approach this problem, we must start by deciding how to specify our computer programs; that is, we must choose a programming language. For reasons described below, this dissertation focuses on a common subset of a family of languages known as *pure functional programming languages*.

What is a pure functional programming language? Most programming languages include some notion of “functions”, although what that means can vary significantly from one programming language to another. Generally a “function” is a piece of code that takes some number of *arguments* as input, and then computes a *result* that is passed back to the code that called it. How well this coincides with the mathematical definition of “function” depends on the programming language.

Many commonly-used programming languages, including C and Java, are *imperative* languages. Functions in imperative languages consist of sequences of *statements*, which are the basic building blocks of algorithms. A simple statement might involve evaluating a mathematical expression that depends on the function arguments. Other statements might have effects beyond just computing a result: They might modify a value stored in memory, or get a character from user input or from a file. So if the same function is called twice with the same arguments, then it might return a different result each time—we would say that such a function is not *pure*.

In contrast, pure functional programming languages like Haskell [PJ03, Bir98] have a focus on *expressions*—which are evaluated without side-effects—rather than statements. In Haskell, all functions are pure: The result of a function depends only on the values of its arguments, and evaluating a function has no observable effect other than just computing the result value. In a pure functional programming language, it makes sense to think of functions as real functions in the mathematical sense: A mathematical relation between argument and result values is sufficient to specify the behavior of a Haskell function.

Using a pure functional language makes a big difference when it comes to reasoning about equivalence of programs. In a C program, we might have a particular function call `add(3,5)` that returns the value `8`. Does this mean that the program expression "`add(3,5)`" is equivalent to the expression "`8`"? Not necessarily—the function `add` might have other side-effects, such as writing to a global variable, printing output to the screen, or writing to a file. So replacing one expression with the other could change the meaning of the surrounding program—we would say that such a function call is not *referentially transparent*. In contrast, every Haskell function call is referentially transparent: Replacing a function call with its return value always yields an equivalent Haskell program—just as in arithmetic, replacing an occurrence of $(3+5)$ with 8 always yields an equivalent mathematical expression. Thus we can reason about Haskell programs just as we might reason about mathematical formulas: Within a Haskell program, we are free to replace function calls with their values, unfold function definitions, or rewrite expressions using algebraic laws satisfied by the relevant functions.

In the sense that they permit equational reasoning, pure functional languages like Haskell are the easiest to work with. But in another sense, the verification problem for Haskell is the most challenging verification problem of all, because it is the most general. Haskell-like languages have several core features that make them useful for embedding other languages, such as higher-order functions (i.e., functions that take other functions as arguments), algebraic datatypes, polymorphism, and recursion [Hud98]. Historically, many of these language features originate with the Iswim language, from Landin's seminal 1966 paper "The Next 700 Programming Languages" [Lan66]. Iswim was designed specifically to be expressive enough to unify a large class of existing programming languages. The expressiveness of Haskell makes it a worthwhile language to study, because if you know how to reason about Haskell, then you know how to reason about many other languages as well.

1.1 INFORMAL REASONING WITH HASKELL

1.1.1 Haskell terms and types

Haskell [PJ03, Bir98] is a general-purpose programming language based on a typed lambda calculus. The full language includes many features (such as type classes) that will not be relevant for most of this dissertation. Accordingly, this section will focus on just a few basic features: functions, the type system, algebraic datatypes, and recursive definitions.

As it is based on the lambda calculus, Haskell has syntax for function application, written "`f x`"; and function abstraction, written "`\x -> t`".[1] Function application associates to the left, so "`f x y`" means "`(f x) y`". In Haskell, functions with symbol names like "`(+)`" use infix syntax, so that "`f x + y`" means "`(+) (f x) y`". Nested abstractions also have special syntax: "`\x y -> t`" is shorthand for "`\x -> \y -> t`".

[1] "\" is Haskell's ASCII approximation of "λ", the usual symbol for function abstraction in the lambda calculus.

```
                                  3 :: Integer
                                (+) :: Integer -> Integer -> Integer
                        (\x -> x + 3) :: Integer -> Integer
                          (\f -> f 3) :: (Integer -> a) -> a
                    (\f x -> f (f x)) :: (a -> a) -> a -> a
                               [3, 5] :: [Integer]
                       (\x -> [x, x]) :: a -> [a]
                      (\xs -> 3 : xs) :: [Integer] -> [Integer]
(\xs -> case xs of [] -> 3; y : ys -> 4) :: [a] -> Integer
```

Figure 1.1: Haskell expressions with types

As a typed language, every expression in a Haskell program belongs to a *type*; we write `x :: T` to assert that expression `x` has type `T`. Haskell types include base types like `Integer`, function types like `Integer -> Integer`, and datatypes like `[Integer]` (read as "list of integer"). (The function arrow is right-associative, so the two-argument function type `A -> B -> C` really means `A -> (B -> C)`, a function that returns a function.) Haskell also includes *polymorphic* types like `a -> a`, which mention type variables like "`a`" (type variables are distinguished from other Haskell types by being in lower-case). A polymorphic type can be *instantiated* by uniformly substituting types for type variables; for example, `Integer -> Integer` is an instance of the polymorphic type `a -> a`. Some examples of Haskell expressions with their types are shown in Figure 1.1.

Haskell programmers can define new types using *datatype declarations*. For example, the standard Haskell library contains the declaration `data Bool = False | True`, which introduces a new type `Bool`, with constructors `False :: Bool` and `True :: Bool`.

A slightly more complicated example is the following binary tree datatype. Note that the type being defined also occurs on the right-hand side, making this a *recursive* datatype.

```
data Tree = Leaf Integer | Node Tree Tree
```

This introduces a new datatype `Tree`, with constructor functions `Leaf :: Integer -> Tree` and `Node :: Tree -> Tree -> Tree`. Inhabitants of type `Tree` include expressions like `Leaf 3` or `Node (Leaf 2) (Node (Leaf 3) (Leaf 4))`.

Datatypes may also have type parameters. Below is a variation of the `Tree` datatype that is parameterized by the type of values contained in the leaves:

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

With this new `Tree'` datatype, the constructor functions now have polymorphic types. This means we can use the same constructor functions to build different trees with elements of different types. For example, we have `Leaf' 5 :: Tree' Integer` and `Leaf' True :: Tree' Bool`.

Haskell's list type is an ordinary datatype, but with some special syntax. It could be defined by the declaration `data [a] = [] | a : [a]`. It has two polymorphic constructors, `[] :: [a]` (called "nil") and `(:) :: a -> [a] -> [a]` (called "cons", which is short for "constructor"). Cons is written infix, and associates to the right. The list syntax "`[x, y, z]`" stands for "`x : y : z : []`". Using pluralized names like "`xs`" or "`ys`" for list variables is a common convention among Haskell programmers; this convention is also followed in this document.

1.1.2 Equational reasoning

The simplest kind of proofs about programs are done by equational reasoning: unfolding definitions, performing reduction steps—in general, just replacing equals by equals. Many properties about programs can be proven correct using equational reasoning alone. For example, by unfolding and refolding the definitions, we can show that the function composition operator `(.)` is associative:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(f . g) x = f (g x)
```

Theorem 1.1.1. *For all* `f`*,* `g`*,* `h`*, and* `x`*,* `((f . g) . h) x = (f . (g . h)) x`*.*

Proof. We proceed by equational reasoning, using the definition of `(.)`.

```
((f . g) . h) x
  = (f . g) (h x)
  = f (g (h x))
  = f ((g . h) x)
  = (f . (g . h)) x
```

□

Properties proved by equational reasoning are easy to trust, because in a pure language like Haskell, replacing equals by equals is universally valid. There are no subtle side conditions or restrictions on when and where you are allowed to perform an equational rewriting step. This means that we can achieve a high level of assurance even for informal pencil-and-paper proofs by equational reasoning.

1.1.3 Proofs by induction

Equational rewriting is not sufficient to prove all properties we might be interested in. Many properties, especially those related to recursive functions, also require the use of induction. For example, we can define a recursive function `map` that applies the given function `f` to every element of a list. Then we can prove that mapping the composition of two functions over a list is the same as mapping one, then mapping the other.

```
map :: (a -> b) -> [a] -> [b]
map f []       = []
map f (x : xs) = (f x) : (map f xs)
```

Theorem 1.1.2. *For any* `f`*,* `g`*, and* `xs`*,* `map (f . g) xs = map f (map g xs)`*.*

We proceed by induction over `xs`. For the base case, we show that the proposition holds for `xs = []`, using the definition of `map`:

```
map (f . g) [] = [] = map f [] = map f (map g [])
```

For the induction step, we assume that the proposition holds for an arbitrary `xs`, and then show that the proposition must also hold for `x : xs`.

```
map (f . g) (x : xs)
  = ((f . g) x) : (map (f . g) xs)    (Unfolding map)
  = (f (g x)) : (map (f . g) xs)      (Unfolding (.))
  = (f (g x)) : (map f (map g xs))    (Inductive hypothesis)
  = map f ((g x) : (map g xs))        (Folding map)
  = map f (map g (x : xs))            (Folding map)
```

This may look like a thorough proof, and in some functional languages it would indeed be a valid proof. But in Haskell, there are some extra side conditions that we must check, due to a certain language feature of Haskell—*laziness*. Lazy functional programming languages are pure functional languages that implement a particular evaluation order. Arguments to functions are not necessarily evaluated before a function is called; instead, the evaluation of each argument is deferred until the point where its value is actually needed. (Conversely, *strict* functional languages evaluate arguments before every function call.) In a lazy language, we must consider non-termination in more contexts than we would in a strict language.

1.1.4 Bottoms and partial values

The presence of laziness and non-termination makes reasoning a bit more complicated. In particular, we will need to extend our induction rule for Haskell lists to explicitly consider non-termination.

$$\frac{P(\texttt{[]}) \qquad \forall\,\texttt{x xs}.\ P(\texttt{xs}) \longrightarrow P(\texttt{x : xs}) \qquad \ldots ?}{\forall\,\texttt{xs}.\ P(\texttt{xs})} \tag{1.1}$$

To illustrate this, we will consider a property of a function that reverses the elements of a list. The implementation of `rev` also uses a helper function `snoc` ("cons" spelled backwards) which adds a single element to the *end* of a list.

```
rev :: [a] -> [a]
rev []       = []
rev (x : xs) = snoc (rev xs) x

snoc :: [a] -> a -> [a]
snoc []       y = y : []
snoc (x : xs) y = x : snoc xs y
```

We might like to prove, for example, that reversing a list twice will give back the original list. Since `rev` is defined in terms of `snoc`, we might start by attempting to prove the following proposition as a lemma:

Proposition 1.1.3. *For all* `xs` *and* `y`*,* `rev (snoc xs y) = y : rev xs`*.*

We proceed by induction over `xs`. For the base case, it is straightforward to show that the proposition holds for `xs = []` (both sides evaluate to `[y]`). The `x : xs` case also goes through just fine. However, the property does not hold in general in Haskell, because it does not hold in the case where the evaluation of `xs` fails to terminate.

We say that a Haskell expression is *undefined* if its evaluation leads to non-termination. Any undefined Haskell expression can be treated as equivalent to the canonical Haskell function `undefined`, which goes into an infinite loop if we ever try to evaluate it. In mathematical notation, the value denoted by `undefined` is written $\perp$ (pronounced “bottom”).

```
undefined :: a
undefined = undefined
```

To cover the possibility of non-termination, we can add a third case to our inductive proof of Theorem 1.1.2, where `xs = undefined`. The proof of this case relies on the fact that `map` is strict in its second argument, i.e. `map f undefined = undefined`. This follows from the fact that evaluating `map f xs` immediately requires `xs` to be evaluated.

```
map (f . g) undefined
  = undefined
  = map f undefined
  = map f (map g undefined)
```

Even after proving the undefined case, we are still not quite done with the proof. There is another, more subtle “admissibility” condition we must verify.

1.1.5 Infinite values and admissibility conditions

In addition to undefined values, laziness also introduces the possibility of infinite values. For example, in Haskell we can define an infinite list of booleans:

```
trues :: [Bool]
trues = True : trues
```

In a strict functional language, this definition would yield `trues = undefined`. However, since the constructor function `(:)` is not strict in Haskell, the circular definition of `trues` is only evaluated as far as required by other functions that examine the list—it does not immediately go into an infinite loop. (Of course, other functions taking `trues` as input might still loop, for example if they try to find the end of the list!)

The presence of infinite lists means that our induction principle for lists will need another side condition. We can demonstrate this need by considering an erroneous proof by induction.

```
take :: Integer -> [a] -> [a]
take n []        = []
take n (x : xs) = if n > 0 then take (n - 1) xs else []
```

Using `take` we can define a finiteness predicate for lists, where `xs` is finite if there exists an integer `n` such that `xs = take n xs`. Now we can write down an inductive “proof” that all lists are finite: All three cases (`undefined`, `[]`, and `x : xs`) are provable. However, the list `trues` does not satisfy the finiteness property; where did the proof go wrong?

It turns out that in lazy functional languages like Haskell, where datatypes may contain infinite values, the induction scheme for lists is only valid for the so-called “admissible” predicates. The map-compose property in Theorem 1.1.2 is admissible, while the finiteness predicate is not. (The definition and properties of admissibility will be covered in depth in Chapter 2.)

Already, we have noticed that proofs involving induction are much more subtle and error-prone than proofs using only equational reasoning. And these are not complicated examples—lists are a simple recursive datatype, and `map` and `(.)` have short definitions. As we move toward larger, more complex definitions that use more interesting forms of recursion, it becomes apparent that pencil-and-paper proofs will no longer be sufficient. To get a reasonable level of confidence, we will need completely formal, machine-checked proofs.

1.2 A PREVIEW OF FORMAL REASONING WITH HOLCF ’11

Traditionally, most mathematics is done with *informal* reasoning: People write proofs, which are checked by having other people read and understand them. Many details may be omitted, as long as enough are included to convey an understanding of the proof. In contrast, *formal* proofs are completely rigorous, and omit nothing. Checking a formal proof does not rely on understanding; rather, it consists of mindlessly checking that every logical inference in the proof is valid. Working with formal proofs by hand is generally impractical; however, computers are perfectly suited to the repetitive tasks of constructing and checking formal proofs.

HOLCF ’11 (usually pronounced “hol-cee-eff”) is a system for doing formal reasoning about functional programs. It is implemented as an extension of Isabelle [NPW02], which is a generic interactive theorem prover, or *proof assistant*.

A proof assistant is a piece of software that lets users state definitions and theorems (expressed in a suitable logical formalism) and create formal proofs. The proof assistant facilitates this task by checking logical inferences, keeping track of assumptions and proof obligations, and automating easy or repetitive subproofs.

Users interact with HOLCF '11 by composing and stepping through a *theory file*. Figure 1.2 shows an example theory file with a formalized version of the map-compose theorem from the previous section. The file starts with a theory name and an **imports** declaration specifying the standard HOLCF theory library. HOLCF '11 provides commands that simulate Haskell-style definitions: **domain** for recursive datatypes and **fixrec** for recursive functions. Users state theorems with the **lemma** command, and prove them by writing proof scripts consisting of one or more *proof tactics*. Isabelle lets users step individually through each **apply** command in a proof script, displaying in a separate output window the remaining subgoals that still need to be proved. When all subgoals have been discharged, Isabelle prints the message, "No subgoals!" At this point the **done** command completes the proof.

HOLCF '11 provides automation so that users can prove many theorems in just one or two steps. In the informal proof of the previous section, the strictness of `map` was established by an appeal to intuition about evaluation order; in HOLCF '11 the corresponding lemma map_strict is rigorously proved in one step with the help of the fixrec_simp tactic (documented in Chapter 3). The informal inductive proof of the map-compose property required a moderate amount of equational reasoning, but the automation in HOLCF '11 lets us prove it in just two steps: First, the induct xs tactic applies the induction rule for the lazy list type, yielding separate subgoals for Nil, Cons, $\bot$, and the admissibility check. The simp_all tactic then uses Isabelle's simplifier to discharge all remaining subgoals by equational rewriting. Because it is declared with the [simp] attribute, the simplifier uses the previous lemma map_strict as a rewrite rule in this proof.

```
theory LazyList imports HOLCF begin

  domain 'a List = Nil | Cons (lazy "'a") (lazy "'a List")

  fixrec map :: "('a → 'b) → 'a List → 'b List"
    where "map·f·Nil = Nil"
    | "map·f·(Cons·x·xs) = Cons·(f·x)·(map·f·xs)"

  lemma map_strict [simp]: "map·f·⊥ = ⊥"
    apply fixrec_simp
    done

  lemma map_map: "map·f·(map·g·xs) = map·(Λ x. f·(g·x))·xs"
    apply (induct xs)
    apply simp_all
    done

end
```

Figure 1.2: A HOLCF '11 theory file containing a formalization of lazy lists

In summary, we can see that using HOLCF '11 has some advantages over informal reasoning. First, note that HOLCF '11 is sufficiently expressive to reason about functional programs: Users can directly specify functional programs and theorems about them, using a notation that is similar to a Haskell-like functional programming language. Second, HOLCF '11 has automation: With its proof tactics, users can write concise proof scripts without having to devote attention to routine proof details. Finally, HOLCF '11 provides confidence: It is built within a completely formal and rigorous theorem proving system, preventing errors and guaranteeing the soundness of the results.

1.3 HISTORICAL BACKGROUND

HOLCF ’11 represents the latest step in a long line of research, starting with the work of Dana Scott and Robin Milner in the late 1960s and early 70s. Their research program started with Scott’s logic of computable functions (LCF), which formed the basis for Milner’s original LCF theorem prover. Since the first version of the LCF system, there have been some notable long-term trends in interactive theorem proving: Proof assistants have gradually become more powerful and more automated; at the same time there has been a continued effort to minimize the amount of code and the number of axioms that must be trusted. For a more in-depth account, see the excellent historical overview by Mike Gordon [Gor00].

1.3.1 Logic of computable functions

The logic of computable functions was designed by Dana Scott in 1969 for reasoning about functional programs [Sco93]. The logic comprises two syntactic classes of entities: *terms* and *formulae*. The language of terms is a typed lambda calculus, similar to Haskell—terms are built from variables, constants, lambda abstraction and function application. In addition to function types, LCF also has types o and ι of truth values and numbers, corresponding to the Haskell types `Bool` and `Integer`. (It is straightforward to extend LCF with additional base types, if desired.) Constants in LCF include the truth values $\mathbf{T} : o$ and $\mathbf{F} : o$, a bottom value $\bot_\alpha : \alpha$ for every type α, a conditional (if-then-else) operator $\mathbf{C}_\alpha : o \to \alpha \to \alpha \to \alpha$, and a fixed point combinator $\mathbf{Y}_\alpha : (\alpha \to \alpha) \to \alpha$ for expressing recursive functions.

The language of LCF formulae includes connectives of first-order logic ($\wedge$, $\longrightarrow$, $\forall$), and also (in)equalities between terms, written $t \sqsubseteq u$ and $t = u$. The logical inference rules for LCF include the usual rules of first-order logic, plus a few axioms about inequalities: $(\sqsubseteq)$ is reflexive, antisymmetric, transitive, has $\bot_\alpha$ as a minimal value, and also satisfies the following monotonicity property.

$$\frac{f \sqsubseteq g \qquad x \sqsubseteq y}{f(x) \sqsubseteq g(y)} \tag{1.2}$$

LCF also axiomatizes a rule for proofs by cases on type o, and defining equations for the conditional operator. For the fixed point operator applied to a function $f : \alpha \to \alpha$, LCF gives us an unfolding rule $f(\mathbf{Y}_\alpha(f)) = \mathbf{Y}_\alpha(f)$ and a fixed-point induction principle:

$$\frac{\Phi[\bot_\alpha] \qquad \forall x.\ \Phi[x] \longrightarrow \Phi[f(x)]}{\Phi[\mathbf{Y}_\alpha(f)]} \tag{1.3}$$

Fixed-point induction is valid for any formula $\Phi[x]$ that satisfies a syntactic admissibility test for the variable x.

The axioms of LCF were not really invented, but rather discovered: The logic was designed with a particular model in mind, and the various LCF axioms came from properties that could be proven about the model [Sco93]. The model of LCF is based on *domain theory*, a field of mathematics also pioneered by Dana Scott [GS90]. Each LCF type is modeled as a *domain* (a kind of complete partial order) and LCF functions are modeled as continuous functions between domains. Domain theory also provides a least fixed-point combinator, used to model $\mathbf{Y}_\alpha$. Keep in mind, however, that while domain theory justifies the rules, it is not part of the formal system; LCF is really just a collection of abstract syntactic rules for manipulating formulae.

1.3.2 LCF style theorem provers

In the early 1970s, a few years after LCF was introduced, a proof assistant was developed for it by Robin Milner at Stanford; this first version was known as Stanford LCF. The prover implemented all the formal rules of the LCF logic, providing a programmable interface with which users could interactively prove theorems.

A few years later while at Edinburgh, Milner created a new version called Edinburgh LCF [GMW79], which was the first proof assistant to be implemented in what became known as the "LCF style". In an LCF style prover, all the code for the logical inference rules is collected in a *proof kernel*, which implements an abstract type `thm` of theorems. For example, one of the kernel functions in Edinburgh LCF implements the modus ponens rule: Given a theorem $P \longrightarrow Q$ and a theorem P', after checking that P and P' are the same formula, it creates a theorem Q. As an abstract type, the representation of a `thm` is not visible to any code outside the kernel; only the kernel can create values of type `thm`. Other code cannot forge theorems, but can only create them via the operations exported by the kernel.

The LCF style requires an implementation language that enforces abstract types. Such languages were not commonplace in the 1970s: In order to build Edinburgh LCF, Milner simultaneously developed the functional language ML for this purpose [GMW79]. ML eventually became a widely-used general purpose programming language, and influenced many later functional programming languages, including Haskell.

In an LCF style theorem prover, users can freely extend the system by adding more code outside the kernel, without risking the soundness of the system. Even if the new user code contains bugs, the worst that can happen is that proofs relying on that code might fail; bugs in non-kernel code cannot be exploited to produce invalid theorems.

The LCF architecture makes it possible to write proof assistants that include very large, sophisticated tools for constructing proofs, yet require only a small amount of trusted code. For example, all versions of LCF have included a simplifier for doing proofs by rewriting with conditional rewrite rules. While the simplifier of Edinburgh LCF was still coded into its proof kernel, its successor Cambridge LCF (developed primarily by Paulson in the early 1980s) took advantage of the

LCF architecture by implementing the simplifier outside the kernel. This yielded a system that was just as powerful and more flexible, yet with a significantly smaller trusted code base [Pau87].

1.3.3 Higher order logic and the definitional approach

The proof assistants that succeeded Cambridge LCF switched from LCF to a new logic—higher order logic, also known as HOL—primarily due to a new focus on hardware-related verification tasks that did not require features of LCF like general recursion or fixed point induction. Multiple lines of provers for HOL have been developed since the late 1980s, including Gordon's HOL series (starting with HOL88 and HOL90, leading up to the modern HOL4) and Paulson's Isabelle/HOL theorem prover.

The syntax and type system of HOL are similar to LCF, but instead of interpreting types as domains, HOL types are modeled as ordinary sets. Accordingly, HOL drops some of LCF's features: There is no special bottom value ($\perp$) at every type, nor is there a generic fixed point combinator. However, HOL has an advantage in expressiveness over LCF, because it is higher order: That is, it can express quantification over formulas and predicates (which in HOL are simply terms with types like bool and 'a $\Rightarrow$ bool).

As new proof assistants started using higher order logic, they also began to shift from an *axiomatic* to a *definitional* approach to building theories. "The HOL system, unlike LCF, emphasises definition rather than axiom postulation as the primary method of developing theories. Higher order logic makes possible a purely definitional development of many mathematical objects (numbers, lists, trees etc.) and this is supported and encouraged." [Gor00, §5.2]

The obvious problem with axioms is that the whole set of them, taken together, must be trusted to be consistent. When users are adding new arbitrary axioms all the time, it is hard to maintain a high level of confidence in the soundness of the system. On the other hand, the definitional approach prescribes certain forms of “safe” axioms for introducing new constants and types, which are known to preserve soundness—users can freely add such definitional axioms without worry.

For defining constants, it is safe to introduce axioms of the form $\mathsf{c} = t$, where c is a new constant and t is a closed term that does not mention c. For example, in Isabelle/HOL, the existential quantifier $\mathsf{Ex} :: (\mathsf{'a} \Rightarrow \mathsf{bool}) \Rightarrow \mathsf{bool}$ is defined by declaring the definition axiom $\mathsf{Ex} = (\lambda \mathsf{P}.\ \forall \mathsf{Q}.\ (\forall \mathsf{x}.\ \mathsf{P}\ \mathsf{x} \longrightarrow \mathsf{Q}) \longrightarrow \mathsf{Q}))$. The standard rules for reasoning about existentials can be derived from this definition. Contrast this with Cambridge LCF, where existential quantification is hard-coded into the formula language, and all the rules about it are axioms. (Note that this definition of Ex could not even be expressed in the first-order LCF, since it has a higher-order type and involves quantification over formulae.)

The safe way to introduce a new type in HOL is to identify values of the new type with some subset of the values of a pre-existing type. (This kind of type definition is discussed further in Chapter 2.) For example, in Isabelle/HOL, the product type $\mathsf{'a} \times \mathsf{'b}$ is not primitive; it is defined in terms of a subset of type $\mathsf{'a} \Rightarrow \mathsf{'b} \Rightarrow \mathsf{bool}$. Each pair (x, y) corresponds to the binary predicate that is true only when applied to x and y, and nowhere else. Compare this with the treatment of pairs in LCF: The properties of the LCF product type and `PAIR` constructor are all given by axioms.

Using these low-level definitional principles for constants and types, it is possible to build high-level derived definition packages. Generally, a definition package takes a user-supplied specification of a type or constant, internally translates it into a low-level definition, and then derives high-level theorems about it. For example, Melham extended HOL88 with a definitional datatype package: Given a (possibly recursive) datatype specification, it would define the type, along with constructor functions and a recursion combinator, and derive an induction rule and other theorems [Mel89]. A similar package was developed for Isabelle/HOL soon afterwards [BW99]. Another classic example is the RECDEF package, which defines functions using well-founded recursion [Sli96]. Here the user supplies a set of recursive function equations and a proof of termination; the package internally creates a non-recursive low-level definition, and then proves the given equations as theorems. Packages like these provide a lot of power and automation to users, yet because they adhere to the definitional approach, they guarantee soundness without users having to trust any new code.

1.3.4 Isabelle/HOLCF

The previous section showed some of the benefits of higher order logic over LCF, particularly the definitional approach for building trustworthy theorem proving systems. But HOL still has one drawback compared to LCF: HOL does not have a general fixed point combinator, so it does not work as well as LCF for reasoning about functional programs with general recursion. Isabelle/HOLCF is the result of an attempt to augment HOL with some features of LCF, so that users can do LCF-style reasoning about LCF terms in Isabelle/HOL.

What exactly is HOLCF? It is not actually a separate logic from HOL, in the sense that LCF and HOL are separate logics. Rather, it is a *model* of the LCF logic, embedded in Isabelle/HOL. To show exactly what it means for one logic to be embedded in another, it may be helpful to consider a much simpler example.

Example: Embedding LTL in HOL. Linear temporal logic (LTL) is a formalism for expressing and reasoning about propositions that depend on time, where time progresses in discrete steps. LTL includes standard logical connectives ($\neg$, $\wedge$, $\vee$) and also some *modal operators*: $\bigcirc P$ means that predicate P holds at the next time step, and $P \,\mathcal{U}\, Q$ means that P holds at every time step until Q becomes true at some point in the future.

In order to reason about LTL propositions, one possibility would be to write an interactive theorem prover that directly implements the LTL logic. The basic logical connectives and modal operators could be implemented as primitives, and each of the logical inference rules for LTL could be coded into the proof kernel. Another alternative is to *embed* LTL inside a more expressive system, such as Isabelle/HOL. To implement the embedding, we fix a model of LTL, and then *define* the LTL connectives in Isabelle/HOL in terms of their meanings in the model.

The usual model for LTL interprets propositions as infinite sequences of truth values, i.e. functions of type nat ⇒ bool. In this model, an LTL proposition is "true" if it is true *now*, i.e. at time zero. Logical connectives of LTL are modeled by combining sequences pointwise; the *next* operator shifts sequences by one.

```
type_synonym ltl_prop = "nat ⇒ bool"

definition TrueLTL :: "ltl_prop ⇒ bool" ("⊨")
  where "⊨ P = P 0"

definition AND :: "ltl_prop ⇒ ltl_prop ⇒ ltl_prop" (infixr "⅄" 55)
  where "(P ⅄ Q) = (λn. P n ∧ Q n)"

definition NEXT :: "ltl_prop ⇒ ltl_prop" ("◯")
  where "◯P = (λn. P (n + 1))"
```

Similarly implementing all of the LTL connectives as definitions makes it possible to express any LTL proposition as a formula in Isabelle/HOL. To support LTL proofs, we can go on to prove each LTL inference rule as a theorem about the model of LTL. For example:

```
lemma AND_intro:
  assumes "⊨ P" and "⊨ Q" shows "⊨ (P ⅄ Q)"

lemma NEXT_AND:
  assumes "⊨ (◯P ⅄ ◯Q)" shows "⊨ (◯(P ⅄ Q))"
```

A theory file could be created with proof scripts for both of these lemmas. LTL proofs could then be replayed in the Isabelle/HOL model of LTL using these rules.

Embedding LCF in HOL. HOLCF is an embedding of LCF in Isabelle/HOL—essentially a formalization of a model of LCF. Each of the base types and type constructors in LCF's type system corresponds to a type definition in Isabelle/HOLCF. Each primitive constant ($\mathbf{T}$, $\mathbf{F}$, $\bot_\alpha$, $\mathbf{C}_\alpha$, $\mathbf{Y}_\alpha$) and term constructor (application and abstraction) in LCF's term language is defined as a constant in Isabelle/HOLCF, allowing any LCF term to be encoded as an Isabelle term. The formula language of LCF is similarly mapped onto Isabelle terms of type bool, so that any LCF formula can be expressed in Isabelle.

As mentioned earlier, the standard model of LCF uses domain theory: LCF types are modeled as complete partial orders (cpos); the function space of LCF is modeled as the continuous function space. Formulae are modeled in the Isabelle/HOL type bool. LCF terms are modeled using domain theory: Each LCF type is modeled in HOL as a type with a cpo structure. The LCF function type constructor is modeled in HOLCF as a new continuous function type constructor, which is separate from the existing Isabelle/HOL function type. Things like admissibility (which is a primitive concept hard-wired into the kernel of the original LCF provers) are defined as predicates in HOLCF.

Versions of HOLCF. The first version of HOLCF (which we will call HOLCF '95) was created in Munich by Regensburger [Reg94, Reg95] in the mid 1990s. In terms of features, HOLCF '95 attempted to precisely replicate the implementation details of Cambridge LCF: In particular it defines all of the same type constructors and operations that were provided by Cambridge LCF.

Over the next few years, HOLCF was extended by various members of the Munich group [MNOS99], resulting in a version we will call HOLCF '99. This version included one new feature in particular that brought big gains in expressiveness and automation, greatly expanding the set of programs that HOLCF could reason about: the DOMAIN package [Ohe97], which provides a high-level datatype definition command for recursive datatypes. It is similar to the datatype packages in Gordon HOL and Isabelle/HOL, except that it defines cpos with bottom values. It automates the same process by which Edinburgh and Cambridge LCF users would axiomatize new datatypes. Unfortunately, the HOLCF '99 DOMAIN package takes a step backward in terms of the trusted code base: Since it is axiomatic rather than definitional, the soundness of HOLCF '99 depends on the correctness of much of the DOMAIN package code.

In the late 2000s, HOLCF was thoroughly revised and extended by the present author. HOLCF '11 is the latest version of HOLCF; it is included as part of the 2011 release of the Isabelle theorem prover.

1.4 THESIS STATEMENT

The original HOLCF '95 consisted of a domain-theoretic model of the basic LCF logic in Isabelle/HOL. HOLCF '99 essentially marked a move from plain LCF to a more expressive logic of LCF+datatypes. However, HOLCF '99 still uses the same domain-theoretic model as HOLCF '95, which does not actually provide meanings for recursive datatypes—axioms are used to fill the gap.

The aim of HOLCF '11 is to take advantage of further developments in domain theory to build a more complete model of LCF, including recursive datatypes. The claim of this thesis is that with the help of some new concepts from domain theory, HOLCF '11 advances the state of the art in formal program verification by offering an unprecedented combination of expressiveness, automation, and confidence.

Expressiveness. HOLCF '11 provides definition packages that allow users to directly formalize a wide variety of functional programs. Compared to earlier versions of HOLCF, the tools in HOLCF '11 have more capabilities, including support for new kinds of function and datatype definitions. The new tools also offer better scalability for larger, more complex datatypes and programs.

Automation. With HOLCF '11, users can verify simple programs in a direct, highly-automated way. Programs that previous systems like HOLCF '99 could verify are now more straightforward to define, and have shorter, more automatic proofs. The improvements in automation also make it possible to complete complicated proofs of theorems for which other reasoning techniques would be impractical.

Confidence. HOLCF '11 provides a strong argument for correctness, because its implementation is *purely definitional*. Our motto: "No new axioms!" HOLCF '11 is a conservative extension of Isabelle/HOL, not requiring a single new line of trusted code.

1.5 OUTLINE

The remainder of this dissertation is organized as follows.

Chapter 2 This covers the formalization of the core parts of HOLCF '11. It includes all of the domain-theoretic concepts and type constructors that were already present in earlier versions of HOLCF. It also describes the CPODEF package that is used to help define types in HOLCF '11, and how automation works for proofs of continuity and admissibility.

Chapter 3 This chapter describes the FIXREC package, which lets users define recursive functions with pattern matching. It covers both usage and implementation.

Chapter 4 This chapter covers the usage and implementation of the DOMAIN package, which is used to define recursive datatypes.

Chapter 5 This documents the HOLCF '11 powerdomain libraries, which are used for reasoning about nondeterministic programs. This chapter also describes the new infrastructure for ideal completion, a general domain-theoretic method for constructing types in HOLCF '11.

Chapter 6 This chapter explains the additions to the HOLCF '11 Domain package that allow it to be purely definitional. The centerpiece of this chapter is the construction of a universal domain—a single cpo with a structure rich enough to encode any recursive datatype.

Chapter 7 The final chapter provides evidence for the claims in the thesis statement, by demonstrating the capabilities of HOLCF '11 on some real examples and making comparisons to related work.

Readers who want to learn to use HOLCF '11, but are not interested so much in the implementation, may want to focus on certain parts of this document. HOLCF '11 users should be able to safely skip the section about Cpodef in Chapter 2, the implementation sections of Chapters 3 and 4, and the second half of Chapter 5 (from ideal completion onwards). Most of Chapter 6 can be skipped, although HOLCF '11 users should at least know about the existence of the domain and predomain type classes (Sections 6.5.4 and 6.7).

The proof methods used in the case studies of Chapter 7 may be of interest, not just to practitioners of formal reasoning in HOLCF '11, but to anyone interested in verification of functional programs in general, formal or otherwise.

Chapter 2

BASIC DOMAIN THEORY IN HOLCF

2.1 INTRODUCTION

Isabelle/HOLCF is a library of domain theory, formalized within the logic of Isabelle/HOL. It is specifically intended to support denotational-style reasoning about recursive functional programs. It is specifically *not* about doing abstract mathematics for its own sake; HOLCF only includes those parts of domain theory that are useful for reasoning about computation.

As was described in the first chapter, HOLCF has undergone various revisions during its history: The original version (HOLCF '95) was created by Regensburger [Reg94, Reg95] as a way to reason about the LCF logic [Pau87] within Isabelle/HOL. The version from a few years later (HOLCF '99) was the next important milestone, representing the work of several contributors [MNOS99]. HOLCF '99 offered improvements to the "core" of HOLCF—i.e., the parts that implemented the basic LCF functionality—and also introduced completely new functionality with the Domain package. The most recent version (HOLCF '11) includes many new improvements and extensions by the present author, covering both the LCF core and the various definition packages. This chapter will describe the new implementation of the core parts of HOLCF '11.

Contributions. Although much of the core of HOLCF '11 is quite similar to HOLCF '99, there are some significant recent improvements as well. The primary original contributions described in this chapter are mostly concerned with proof automation:

- Using *compactness*, admissibility can now be proven automatically for a larger set of predicates.
- The new CPODEF package greatly reduces the burden of constructing new cpo types.
- New tactics for continuity proofs make interactive reasoning faster and feasible for larger programs.

Overview. The remainder of this chapter starts by formalizing abstract concepts from domain theory, such as complete partial orders, continuity and admissibility (§2.2). The following two sections are devoted to the concrete instantiations of these concepts: After introducing a new type definition package for cpos (§2.3), we define several type constructors, along with related operations for each (§2.4). Next is a discussion of proof automation for continuity (§2.5), followed by an evaluation and comparison with related work (§2.6).

Notation. Many definitions and theorems in this document are presented using Isabelle syntax, which is typeset in a sans-serif font. Isabelle generally uses standard mathematical notation for operators and logical connectives. However, due to Isabelle's distinction between object-logic and meta-logic, there are two ways to write some propositions: $\mathsf{P} \longrightarrow \mathsf{Q}$ or $\mathsf{P} \Longrightarrow \mathsf{Q}$ for implication, $\forall \mathsf{x}.\ \mathsf{t}$ or $\bigwedge \mathsf{x}.\ \mathsf{t}$ for universal quantification, and $\mathsf{x} = \mathsf{y}$ or $\mathsf{x} \equiv \mathsf{y}$ for equality. In general, readers can safely ignore the distinction between object- and meta-logic connectives. The syntax $[\![\mathsf{P};\ \mathsf{Q}]\!] \Longrightarrow \mathsf{R}$ represents the nested implication $\mathsf{P} \Longrightarrow \mathsf{Q} \Longrightarrow \mathsf{R}$ (logical implication associates to the right). The bi-implication $\mathsf{P} \longleftrightarrow \mathsf{Q}$ is alternative syntax for $\mathsf{P} = \mathsf{Q}$ on booleans.

Function application in Isabelle uses the syntax $\mathsf{f\ x}$, like in Haskell or ML; application associates to the left, so $\mathsf{f\ x\ y}$ denotes $(\mathsf{f\ x})\ \mathsf{y}$. Function abstraction uses lambda notation $(\lambda x.\ t)$ and may be nested: $(\lambda x\ y.\ t)$ is shorthand for $(\lambda x.\ \lambda y.\ t)$. Function abstraction is the only form of variable binding in Isabelle: Other binding constructs like $(\forall x.\ t)$ are really abbreviations for terms like $\mathsf{All}\ (\lambda x.\ t)$, where All is a higher-order function of type $(\text{'a} \Rightarrow \mathsf{bool}) \Rightarrow \mathsf{bool}$.

Type constructors are generally written postfix, as in int list, although some type constructors have infix syntax, like $\mathsf{int} \times \mathsf{int}$ for the product type or $\mathsf{int} \Rightarrow \mathsf{int}$ for the function space (both type constructors group to the right). Type variables are distinguished by a leading tick mark, as in 'a.

For introducing new types, the **type_synonym** command introduces type abbreviations, and **datatype** defines inductive datatypes, much like in Haskell or ML. Non-recursive constant definitions may be introduced with **definition**; the **primrec** command defines primitive-recursive functions over datatypes.

In Isabelle theory files, theorems to be proved are stated using the **lemma** command. The command is followed by a theorem name, an optional list of theorem attributes (for example, the [simp] attribute adds the theorem to the simplifier), and a quoted proposition.

```
lemma example_theorem [simp]:
  "0 < y ⟹ (x::int) < x + y"
```

Isabelle also supports an alternative form with explicit fixed variables and named assumptions:

```
lemma example_theorem [simp]:
  fixes x :: "int" assumes pos: "0 < y" shows "x < x + y"
```

In Isabelle, **theorem** is a synonym for **lemma**. In this document, we use **lemma** only to refer to theorems proved in the HOLCF '11 library of theory files; **theorem** is reserved for theorems that are generated by a definition package. Isabelle also provides the **have** command for stating sub-lemmas within larger proof scripts; this notation indicates intermediate theorems that packages or tools prove internally but do not export to the user.

In this document, HOLCF '11 library lemmas (presented with the **lemma** keyword) are typically shown without the accompanying formal proof scripts, although informal proof sketches are given for selected lemmas. Complete formal proof scripts for all such lemmas can be found in theory files in the `HOLCF` directory of the Isabelle 2011 distribution.

2.2 ABSTRACT DOMAIN THEORY

This section describes the HOLCF formalizations of various abstract concepts from domain theory. (The HOLCF '11 definitions shown in this section are virtually unchanged since HOLCF '99, and most date back to HOLCF '95.) We start with partial orders, chains, and completeness, working toward the final goal of this section: the fixed point combinator and fixed point induction.

2.2.1 Type class hierarchy for cpos

Every version of HOLCF defines various classes of orders using Isabelle's type class mechanism [Haf10]. A type class is a way to formalize an algebraic structure, which consists of some number of operations on a type together with axioms or laws that the operations must satisfy. For example, a class for groups might fix a binary operation, a unary inverse operation, and a zero element, and assume a class axiom for each of the usual laws for groups. Type classes are defined in Isabelle with the **class** command, which works much like the type class mechanism in Haskell. Each new class can derive from any number of superclasses, overloaded constants are specified with the **fixes** option, and class axioms are specified with the **assumes** option.

All of the HOLCF '11 class definitions, along with the definitions of related constants, are shown in Fig. 2.1. We start with a class below that fixes an ordering relation, but assumes nothing about it. Class po adds the usual axioms of a partial order. The is_ub and is_lub relations, defined for class po, are the usual notions of upper bound and least upper bound of a set. In partial orders, least upper bounds are unique (if they exist), which justifies defining the lub function using the unique choice operator. HOLCF also defines syntax (⨆i. Y i) to represent lub (range (λi. Y i)).

Next we define ascending countable chains, and define class cpo with the assumption that every chain has a least upper bound. Note that HOLCF differs from some presentations of domain theory by using countable chains rather than directed sets. This is a reasonable choice for HOLCF, because directed-completeness is actually stronger than what is needed to construct least fixed points; also, being more concrete, chains are often easier to work with. (See [AJ94, §2.2.4] for more discussion related to this design choice.)

We define class pcpo as a cpo where there exists a minimal element; the constant ⊥ is then defined by unique choice.[1]

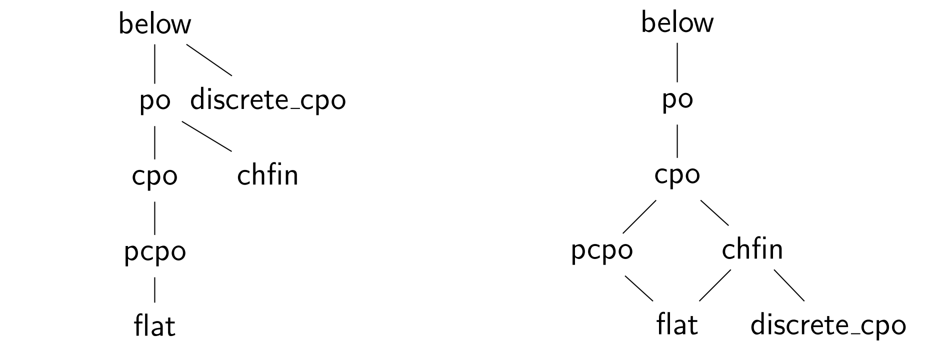
HOLCF '11 also defines three other classes of cpos. The chain-finite types (class chfin) are partial orders where every chain is eventually constant. Chain-finite types were already present in Edinburgh LCF [GMW79] (known there as "easy" types) and in HOLCF '95; they are important for reasoning about admissibility (see Sec. 2.2.3). HOLCF '99 introduced class flat for pointed cpos with a flat ordering, where every non-bottom value is maximal. Types in the new HOLCF '11 class discrete_cpo have a discrete ordering.

Isabelle's class mechanism lets us prove additional subclass relationships beyond the ones declared in the class definitions [Haf10, §3.5]. Accordingly, we prove

[1] Making ⊥ a parameter of class pcpo (using **fixes**) would also be a reasonable design—similarly for the lub constant in the cpo class. The reason for using unique choice is historical: At the time of HOLCF '95 and HOLCF '99, the class mechanism did not support introducing constants and adding axioms about them simultaneously. Using unique choice was necessary for the ⊥ and lub constants to have the right type class constraints.

```
class below =
  fixes below :: "'a ⇒ 'a ⇒ bool" (infix "⊑" 50)

class po = below +
  assumes below_refl: "x ⊑ x"
  assumes below_trans: "⟦x ⊑ y; y ⊑ z⟧ ⟹ x ⊑ z"
  assumes below_antisym: "⟦x ⊑ y; y ⊑ x⟧ ⟹ x = y"

definition is_ub :: "('a::po) set ⇒ 'a ⇒ bool" (infix "<|" 55)
  where "S <| x = (∀y∈S. y ⊑ x)"

definition is_lub :: "('a::po) set ⇒ 'a ⇒ bool" (infix "<<|" 55)
  where "S <<| x = (S <| x ∧ (∀u. S <| u ⟶ x ⊑ u))"

definition lub :: "('a::po) set ⇒ 'a"
  where "lub S = (THE x. S <<| x)"

definition chain :: "(nat ⇒ 'a::po) ⇒ bool"
  where "chain Y = (∀i. Y i ⊑ Y (Suc i))"

class cpo = po +
  assumes cpo: "chain Y ⟹ ∃x. range Y <<| x"

class pcpo = cpo +
  assumes least: "∃x. ∀y. x ⊑ y"

definition bottom :: "'a::pcpo"  ("⊥")
  where "⊥ = (THE x. ∀y. x ⊑ y)"

class chfin = po +
  assumes chfin: "chain Y ⟹ ∃i. ∀j. i ≤ j ⟶ Y i = Y j"

class flat = pcpo +
  assumes ax_flat: "x ⊑ y ⟹ x = ⊥ ∨ x = y"

class discrete_cpo = below +
  assumes discrete_cpo [simp]: "x ⊑ y ⟷ x = y"
```

Figure 2.1: HOLCF '11 type class definitions



(a) subclass relations as defined

(b) subclass relations as proved

Figure 2.2: Type class hierarchy of HOLCF '11

that every chain-finite type is a cpo, and that flat and discrete types are chain-finite. By proving these new subclass relationships, we change the class hierarchy as shown in Fig. 2.2.

2.2.2 Continuous functions

A function between partial orders is *monotone* if it preserves the ordering relation. A function between cpos is *continuous* if it preserves least upper bounds of chains.[2]

```
definition monofun :: "('a::po ⇒ 'b::po) ⇒ bool"
  where "monofun f = (∀x y. x ⊑ y ⟶ f x ⊑ f y)"

definition cont :: "('a::cpo ⇒ 'b::cpo) ⇒ bool"
  where "cont f = (∀Y. chain Y ⟶ range (λi. f (Y i)) <<| f (⨆i. Y i))"
```

We prove some standard theorems relating these two concepts. For example, continuity implies monotonicity; this gives us another useful elimination rule for the cont predicate. A direct corollary is that continuous functions map chains to chains.

```
lemma cont2contlubE:
  "⟦cont f; chain Y⟧ ⟹ f (⨆i. Y i) = (⨆i. f (Y i))"
```

[2] In domain theory, this form of continuity is also known as *ω-continuity* [AJ94]. It may be contrasted with the slightly stronger condition of *directed continuity*, which requires a function to preserve lubs of not just countable chains, but all directed sets.

```
lemma cont2monofunE:
  "⟦cont f; x ⊑ y⟧ ⟹ f x ⊑ f y"

lemma ch2ch_cont:
  "⟦cont f; chain Y⟧ ⟹ chain (λi. f (Y i))"
```

When proving continuity of a function, it is often easier to prove monotonicity first, rather than attempting to prove continuity directly. Therefore HOLCF '11 provides the following introduction rule for the cont predicate for convenience:

```
lemma contI2:
  "⟦monofun f; ⋀Y. ⟦chain Y; chain (λi. f (Y i))⟧ ⟹ f (⨆i. Y i) ⊑ (⨆i. f (Y i))⟧
    ⟹ cont f"
```

The identity and constant functions are clearly continuous. The rule cont_apply shows continuity of a larger function from continuity of its subterms; cont_compose follows as a special case. Lemma cont2cont_lub essentially says that the lub of a chain of continuous functions is itself continuous. The proofs of these last rules use contI2, along with the elimination rules (with names ending in E) for cont listed above.

```
lemma cont_id: "cont (λx. x)"

lemma cont_const: "cont (λx. c)"

lemma cont_apply:
  "⟦cont t; ⋀x. cont (λy. f x y); ⋀y. cont (λx. f x y)⟧ ⟹ cont (λx. (f x) (t x))"

lemma cont_compose:
  "⟦cont c; cont f⟧ ⟹ cont (λx. c (f x))"

lemma cont2cont_lub:
  "⟦⋀x. chain (λi. F i x); ⋀i. cont (λx. F i x)⟧ ⟹ cont (λx. ⨆i. F i x)"
```

The subclasses of cpos defined in the previous section provide some shortcuts for proving continuity. All monotone functions with chain-finite domain are continuous, as are all strict functions with flat domain. Furthermore, every function with a discrete domain is continuous. Each of the next few lemmas is constrained to particular subclasses of cpos, as indicated by the class annotations on each type variable.

```
lemma chfindom_monofun2cont:
  "monofun f ⟹ cont (f::('a::chfin) ⇒ ('b::cpo))"

lemma flatdom_strict2cont:
  "f ⊥ = ⊥ ⟹ cont (f::('a::flat) ⇒ ('b::pcpo))"

lemma cont_discrete_cpo:
  "cont (f::('a::discrete_cpo) ⇒ ('b::cpo))"
```

In addition to the cont predicate, HOLCF also defines a type 'a → 'b of continuous functions, which coexists with Isabelle's ordinary set-theoretic function space type 'a ⇒ 'b. Application and abstraction of continuous functions use the special HOLCF syntax f·x and Λ x. t, compared with f x and λx. t for the ordinary function space. Details about the definition of the continuous function type are found in Sec. 2.4.3.

2.2.3 Fixed points, admissibility, and compactness

It is a well known fact in domain theory that any continuous function $f : D \to D$ over a pointed cpo D has a least fixed point. Furthermore, the least fixed point can be constructed by starting with $\bot$, iterating f, and taking the least upper bound: $\mathit{fix}(f) = \bigsqcup_{n\in\omega} f^n(\bot)$.

We formalize this construction in HOLCF '11 by defining the operations iterate and fix as shown below. Note the use of the continuous function space type 'a → 'a, which ensures that fix can only be applied to continuous functions.

```
primrec iterate :: "nat ⇒ ('a::cpo → 'a) → ('a → 'a)"
  where "iterate 0 = (Λ f x. x)"
  | "iterate (Suc n) = (Λ f x. f·(iterate n·f·x))"

definition fix :: "('a → 'a) → 'a::pcpo"
  where "fix = (Λ f. ⨆n. iterate n·f·⊥)"
```

The fact that fix·f is indeed a least fixed point of f is formalized in the following two lemmas. Informal proofs of these properties can be found in any domain theory textbook; formal proof scripts reside in the theory file `Fix.thy` of the HOLCF '11 distribution.

```
lemma fix_eq: "fix·f = f·(fix·f)"
lemma fix_least_below: "f·x ⊑ x ⟹ fix·f ⊑ x"
```

For proving properties about least fixed points, we formalize the principle of fixed point induction and the associated notion of *admissibility*. Fixed point induction is a primitive rule in the LCF logic; the admissible predicates are those for which the fixed point induction principle is valid. In the Edinburgh and Cambridge LCF systems the notion of admissibility is hard-wired into the kernel as a syntactic check on formulas [GMW79, Pau87]. In contrast, for HOLCF we formalize admissibility in a semantic way, as a higher-order predicate on predicates: A predicate P is admissible if it holds for the least upper bound of a chain whenever it holds for all elements of the chain.

```
definition adm :: "('a::cpo ⇒ bool) ⇒ bool"
  where "adm P = (∀Y. chain Y ⟶ (∀i. P (Y i)) ⟶ P (⨆i. Y i))"
```

Now we can use admissibility to formulate the fixed point induction rule fix_ind.

```
lemma fix_ind: "⟦adm P; P ⊥; ⋀x. P x ⟹ P (f·x)⟧ ⟹ P (fix·f)"
```

Examples using fixed point induction for reasoning about functional programs can be found in the literature [GH05]. Although fixed point induction is a rather low-level form of reasoning, the fixed point induction rule can also be used to derive other higher-level induction principles, such as structural induction [Pau84].

Automation for admissibility proofs. Admissibility conditions arise often in users' proofs, whenever they use fixed point induction, or any other induction rule derived from it. Fortunately, admissibility can be proven automatically for many

lemma adm_const [simp]: "adm (λx. t)"

lemma adm_all [simp]: "⟦⋀y. adm (λx. P x y)⟧ ⟹ adm (λx. ∀y. P x y)"

lemma adm_conj [simp]: "⟦adm P; adm Q⟧ ⟹ adm (λx. P x ∧ Q x)"

lemma adm_disj [simp]: "⟦adm P; adm Q⟧ ⟹ adm (λx. P x ∨ Q x)"

lemma adm_imp [simp]: "⟦adm (λx. ¬ P x); adm Q⟧ ⟹ adm (λx. P x ⟶ Q x)"

lemma adm_iff [simp]:
 "⟦adm (λx. P x ⟶ Q x); adm (λx. Q x ⟶ P x)⟧ ⟹ adm (λx. P x ⟷ Q x)"

lemma adm_below [simp]: "⟦cont u; cont v⟧ ⟹ adm (λx. u x ⊑ v x)"

lemma adm_eq [simp]: "⟦cont u; cont v⟧ ⟹ adm (λx. u x = v x)"

lemma adm_not_below [simp]: "⟦cont t⟧ ⟹ adm (λx. t x ⋢ u)"

Figure 2.3: Simplification rules for admissibility predicate

predicates.

One way to prove admissibility of a predicate is to use a property of the type of the predicate. For chain-finite types, the least upper bound of a chain is always an element of the chain; this means that every predicate on such a type is trivially admissible.

lemma adm_chfin [simp]: "adm (λ(x::'a::chfin). P x)"

Another way is to derive admissibility of a formula from properties of subformulas, using various structural rules. For example, admissibility is preserved by conjunction, disjunction, and universal quantification. Equalities and order comparisons between continuous functions are also admissible. There is no general rule for negation, but negated comparisons like t ⋢ u can be shown to be admissible in x if x does not occur free in u. A set of these admissibility rules is shown in Fig. 2.3; these rules are already present in HOLCF '99 [Mül98, MNOS99].

```
lemma adm_compact_not_below [simp]:
  "⟦compact k; cont t⟧ ⟹ adm (λx. k ⋢ t x)"

lemma adm_neq_compact [simp]: "⟦compact k; cont t⟧ ⟹ adm (λx. t x ≠ k)"

lemma adm_compact_neq [simp]: "⟦compact k; cont t⟧ ⟹ adm (λx. k ≠ t x)"
```

Figure 2.4: Admissibility rules involving compactness

There are some important predicates not covered by the rules in Fig. 2.3; specifically, there is a lack of rules for negated equalities and comparisons. To cover the missing cases, we formalize the standard domain-theoretic concept of a *compact* element. (Compactness is new to HOLCF '11; it was not present in HOLCF '99 or earlier versions.) Intuitively, a compact element of a cpo is one that cannot be approximated by values strictly below itself. More precisely, if x is compact, then for any chain Y where $x \sqsubseteq (\bigsqcup i.\ Y\ i)$, there must exist some element of the chain such that $x \sqsubseteq Y\ i$. Equivalently, we can define compactness in terms of admissibility:

```
definition compact :: "'a::cpo ⇒ bool"
  where "compact k = adm (λx. k ⋢ x)"
```

Using the definition of compactness together with the following substitution property of admissibility [MNOS99], we can derive the additional rules shown in Fig. 2.4.

```
lemma adm_subst: "⟦cont f; adm P⟧ ⟹ adm (λx. P (f x))"
```

All elements of chain-finite types are compact, as is the bottom element of any pointed cpo. Furthermore, we shall see in Sec. 2.4 that for each of the types defined in HOLCF '11, all of the constructor functions preserve compactness.

```
lemma compact_chfin [simp]: "compact (x::'a::chfin)"

lemma compact_bottom [simp]: "compact ⊥"
```

Finally, let us consider a few examples of predicates whose admissibility can be proved automatically. Admissibility of $(\lambda x.\ f{\cdot}x \neq \bot \wedge (\forall y.\ g{\cdot}x = h{\cdot}y))$ can be proven by the simplifier, as can the admissibility of $(\lambda x.\ p{\cdot}x = a \longrightarrow f{\cdot}x \sqsubseteq x)$, as long as a is compact. The predicate $(\lambda x.\ f{\cdot}x \sqsubseteq a \longleftrightarrow g{\cdot}x \sqsubseteq b)$ is always admissible, but $(\lambda x.\ a \sqsubseteq f{\cdot}x \longleftrightarrow b \sqsubseteq g{\cdot}x)$ is only admissible if a and b are compact.

2.3 DEFINING CPOS AS SUBTYPES: THE CPODEF PACKAGE

One way to create instances of the cpo class is by a manual process: Starting with an existing HOL type or datatype, we define an ordering relation, and then proceed to prove that the ordering is a complete partial order. For some types (e.g. cartesian products and function spaces) this is not too difficult, but in general, proving the cpo axioms manually for a new type can be a lot of work, requiring lengthy proof scripts. In addition, defining operations on the new cpo requires manual proofs of continuity, which can also be tedious.

Another way to construct a new cpo is to carve out a subset of an existing cpo; then the new type can inherit the order structure from the old one. Continuous operations on the new type can also be derived from continuous functions on the old cpo. In HOLCF '11, the new Cpodef package automates this type definition process. Several types in HOLCF '11 are now defined using the Cpodef package, including the continuous function space (§2.4.3), the flat domain type 'a lift (§2.4.5), the strict product (§2.4.6), and the strict sum (§2.4.7).

The Cpodef package adds two new commands to HOLCF '11, **cpodef** and **pcpodef**, for defining types in classes cpo and pcpo, respectively. They are implemented by layering new functionality on top of Isabelle's existing **typedef** command [NPW02, §8.5].

Using **typedef** is the most basic way to define a new type in Isabelle; it introduces a new type isomorphic to a nonempty subset of an existing type. As an example, we can define a new type isomorphic to the set of all odd integers:

```
typedef oddint = "{x::int. odd x}"
```

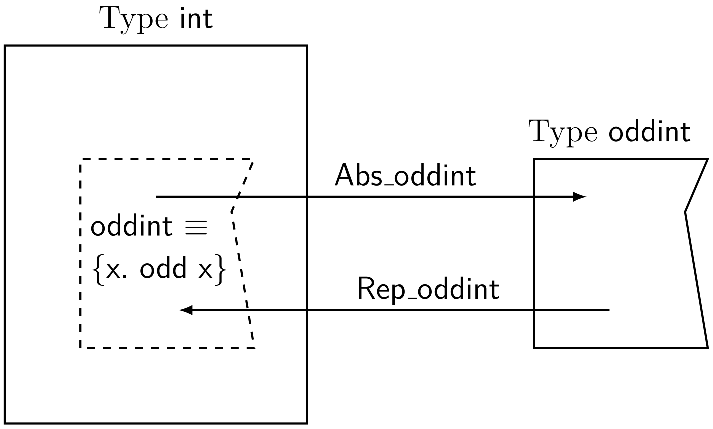


Figure 2.5: Type definition with **typedef**, yielding Rep and Abs functions

After discharging a proof obligation that the given set {x::int. odd x} is nonempty, the Typedef package generates a few constants and theorems. The set oddint is defined equal to the set we specified; we also get a pair of *representation* and *abstraction* functions Rep_oddint and Abs_oddint that map between the set and the new type as shown in Fig. 2.5. The essential properties of these constants are listed below.

```
theorem oddint_def: "oddint ≡ {x::int. odd x}"
theorem Rep_oddint: "Rep_oddint x ∈ oddint"
theorem Rep_oddint_inverse: "Abs_oddint (Rep_oddint x) = x"
theorem Abs_oddint_inverse: "y ∈ oddint ⟹ Rep_oddint (Abs_oddint y) = y"
```

We can use Rep_oddint and Abs_oddint to define new operations on type oddint, like the function oddmult below. Properties like associativity of oddmult can then be proved using theorems provided by Typedef.

```
definition oddmult :: "oddint ⇒ oddint ⇒ oddint"
  where "oddmult x y = Abs_oddint (Rep_oddint x ∗ Rep_oddint y)"
```

All of the generated theorems about Rep_oddint and Abs_oddint are derived from a single axiom of the form "type_definition Rep_oddint Abs_oddint oddint", where the type_definition predicate is defined as follows.[3]

```
definition type_definition :: "('b ⇒ 'a) ⇒ ('a ⇒ 'b) ⇒ 'a set ⇒ bool"
  where "type_definition Rep Abs A ≡
  (∀x. Rep x ∈ A) ∧ (∀x. Abs (Rep x) = x) ∧ (∀y. y ∈ A ⟶ Rep (Abs y) = y)"
```

The Typedef package includes a small library of lemmas, each with an assumption of the form "type_definition Rep Abs A". By instantiating these with a particular type definition axiom, Typedef generates all the theorems about those particular Rep and Abs functions.

The Cpodef package includes a similar library of lemmas with type_definition assumptions. Some of these lemmas are used to derive the theorems that the package generates; others are used to discharge class instance proofs.

Proving a subtype is a partial order. The first lemma in the Cpodef library is typedef_po, which is used for proving po class instances.

```
lemma typedef_po:
  fixes Rep :: "'b::below ⇒ 'a::po" and Abs :: "'a ⇒ 'b" and A :: "'a set"
  assumes type: "type_definition Rep Abs A"
  assumes below_def: "(op ⊑) ≡ (λx y. Rep x ⊑ Rep y)"
  shows "OFCLASS('b, po_class)"
```

The conclusion of the lemma is a special predicate stating that type 'b satisfies all the axioms of class po; it is precisely the proof obligation that users are faced with when they instantiate class po with Isabelle's **instance** command. For class po, we must show that (op ⊑) is reflexive, transitive, and antisymmetric. Reflexivity and transitivity on type 'b follow directly from the same properties on type 'a, after unfolding below_def. Proving antisymmetry also requires injectivity of Rep, which follows from assumption type.

[3] Actually, it is defined with the **locale** command, not with **definition**; but the distinction is not important.

Proving a subtype is a cpo. The next few lemmas in the CPODEF library are related to completeness and continuity. Note that these lemmas have some additional type class constraints on types 'a and 'b. Each lemma also has a new assumption adm_A that asserts the chain-completeness of set A. The first lemma, typedef_is_lub, shows how to construct least upper bounds for chains on type 'b.

```
lemma typedef_is_lub:
  fixes Rep :: "'b::po ⇒ 'a::cpo" and Abs :: "'a ⇒ 'b" and A :: "'a set"
  assumes type: "type_definition Rep Abs A"
  assumes below_def: "(op ⊑) ≡ (λx y. Rep x ⊑ Rep y)"
  assumes adm_A: "adm (λx. x ∈ A)"
  shows "chain Y ⟹ range Y <<| Abs (⨆i. Rep (Y i))"
```

Note that in order to show that any particular x :: 'b is the least upper bound of a chain Y, it suffices to show that Rep x is the least upper bound of Rep mapped over Y. Accordingly, we will show range (λ i. Rep (Y i)) <<| Rep (Abs (⨆i. Rep (Y i))). From type, we know that every Rep (Y i) ∈ A; together with adm_A we then have that (⨆i. Rep (Y i)) ∈ A. In turn, this implies that Rep (Abs (⨆i. Rep (Y i))) equals (⨆i. Rep (Y i)), so our goal simplifies to range (λ i. Rep (Y i)) <<| (⨆i. Rep (Y i)). This final goal is true because (λi. Rep (Y i)) is a chain on a cpo type; this concludes the proof.

From lemma typedef_is_lub, it is a simple matter to derive the lemma typedef_cpo, which shows OFCLASS('b, cpo_class) from the same assumptions. Another direct corollary of typedef_is_lub is typedef_thelub, which concludes that (⨆i. Y i) equals Abs (⨆i. Rep (Y i)) for any chain Y.

Continuity of Rep and Abs. The next pair of lemmas show that the Rep and Abs functions are both continuous.

```
lemma typedef_cont_Rep:
  fixes Rep :: "'b::cpo ⇒ 'a::cpo" and Abs :: "'a ⇒ 'b" and A :: "'a set"
  assumes type: "type_definition Rep Abs A"
  assumes below_def: "(op ⊑) ≡ (λx y. Rep x ⊑ Rep y)"
  assumes adm_A: "adm (λx. x ∈ A)"
  shows "cont Rep"
```

The proof of typedef_cont_Rep starts by applying the standard rule for continuity. We must show that range (λ i. Rep (Y i)) <<| Rep (⨆i. Y i) for an arbitrary chain Y. After substituting lemma typedef_thelub, our proof obligation becomes range (λ i. Rep (Y i)) <<| Rep (Abs (⨆i. Rep (Y i))), which we have already established while proving typedef_is_lub.

The formulation of the continuity rule for Abs is a little different than the rule for Rep. Because the behavior of Abs is unspecified on inputs not in A, we cannot prove that Abs is always continuous; instead we prove that Abs is continuous when composed with another continuous function f whose range is a subset of A.

```
lemma typedef_cont_Abs:
  fixes Rep :: "'b::cpo ⇒ 'a::cpo" and Abs :: "'a ⇒ 'b" and A :: "'a set"
  assumes type: "type_definition Rep Abs A"
  assumes below_def: "(op ⊑) ≡ (λx y. Rep x ⊑ Rep y)"
  assumes adm_A: "adm (λx. x ∈ A)"
  shows "(⋀x. f x ∈ A) ⟹ cont f ⟹ cont (λx. Abs (f x))"
```

The proof of typedef_cont_Abs is by unfolding the definitions of continuity and least upper bounds, and simplifying with the rule Rep (Abs (f x)) = f x (derived from type and the assumption about f.) The assumption adm_A is actually not necessary, but the implementation of CPODEF is simpler when all the lemmas have the same set of assumptions.

Compactness. We prove that an element x :: 'b is compact whenever Rep x is.

```
lemma typedef_compact:
  fixes Rep :: "'b::cpo ⇒ 'a::cpo" and Abs :: "'a ⇒ 'b" and A :: "'a set"
  assumes type: "type_definition Rep Abs A"
  assumes below_def: "(op ⊑) ≡ (λx y. Rep x ⊑ Rep y)"
  assumes adm_A: "adm (λx. x ∈ A)"
  shows "compact (Rep k) ⟹ compact k"
```

We assume compact (Rep k), which by definition means adm (λx. Rep k $\not\sqsubseteq$ x). Then using the continuity of Rep, lemma adm_subst gives us adm (λx. Rep k $\not\sqsubseteq$ Rep x). Using below_def, we see that this is equivalent to adm (λx. k $\not\sqsubseteq$ x), which is the definition of compact k.

Proving a subtype is pointed. The last few theorems in the Cpodef library are about bottoms and the pcpo class. If the defining set A contains $\bot$, then the type 'b will have a least element.

```
lemma typedef_pcpo:
  fixes Rep :: "'b::cpo ⇒ 'a::pcpo" and Abs :: "'a ⇒ 'b" and A :: "'a set"
  assumes type: "type_definition Rep Abs A"
  assumes below_def: "(op ⊑) ≡ (λx y. Rep x ⊑ Rep y)"
  assumes bottom: "⊥ ∈ A"
  shows "OFCLASS('b, pcpo_class)"
```

The proof is straightforward, and involves showing that Abs $\bot$ is the minimal element of type 'b. Finally we proceed to prove lemmas Rep_strict and Abs_strict, which respectively conclude Rep $\bot = \bot$ and Abs $\bot = \bot$ from the same assumptions, using similar reasoning.

Implementing the commands. The **cpodef** and **pcpodef** commands each have an input syntax identical to the **typedef** command, but they differ in the proof obligations they require of the user. Figure 2.6 shows the precise proof obligations required by each tool. Additionally, each command has different class requirements on the representation type. For example, **cpodef** can only be used to define sub-cpos of types that are already in class cpo.

When the **cpodef** command is used, the Cpodef package first obtains proofs of $\exists$x. x $\in$ A and adm (λx. x $\in$ A) from the user. The tool then uses the nonemptiness proof to call the **typedef** command internally, yielding a type_definition theorem. The tool then instantiates class below by defining (op $\sqsubseteq$) $\equiv$ (λx y. Rep x $\sqsubseteq$ Rep y)

Command	Type class	Proof obligation
typedef	type	$\exists$x. x $\in$ A
cpodef	cpo	($\exists$x. x $\in$ A) $\wedge$ adm (λx. x $\in$ A)
pcpodef	pcpo	$\perp$ $\in$ A $\wedge$ adm (λx. x $\in$ A)

Figure 2.6: Comparison of **typedef**, **cpodef**, and **pcpodef** commands

on the new type, and uses lemma typedef_po to prove an instance for class po. Next, it uses the admissibility proof with lemma typedef_cpo to prove the cpo class instance. Finally, it instantiates lemmas like typedef_cont_Rep, typedef_cont_Abs and typedef_compact, and binds them to new type-specific theorem names. More details about the generated theorems can be found in the upcoming sections.

The **pcpodef** command starts by obtaining a proof of $\perp$ $\in$ A $\wedge$ adm (λx. x $\in$ A) from the user. From $\perp$ $\in$ A, the tool derives $\exists$x. x $\in$ A, which is used to call **cpodef** internally; this yields type_definition and below_def theorems, and a cpo class instance. Next it can use lemma typedef_pcpo to prove the pcpo class instance. Finally, it instantiates the strictness theorems for Rep and Abs and gives them type-specific theorem names (see Sec. 2.4.6 for a specific example).

2.4 HOLCF TYPES

HOLCF provides types corresponding to all of the basic constructions of domain theory, with instances of the cpo and pcpo classes, as appropriate. The remainder of this section is devoted to the definitions, operations, and properties of these types as formalized in HOLCF '11: cartesian product (§2.4.1), full function space (§2.4.2), continuous function space (§2.4.3), lifted cpo (§2.4.4), discrete cpos and flat lifted types (§2.4.5), strict product (§2.4.6), and strict sum (§2.4.7).

2.4.1 Cartesian product cpo

If D and E are cpos, then the cartesian product $D \times E$ also forms a cpo, with the componentwise ordering: $(x_1, y_1) \sqsubseteq (x_2, y_2)$ if and only if $x_1 \sqsubseteq x_2 \wedge y_1 \sqsubseteq y_2$. Least upper bounds are also computed componentwise. If D and E both have bottom elements, then $D \times E$ has a minimal element $\bot = (\bot, \bot)$.

In HOLCF '11, as with earlier versions of HOLCF, the product cpo is formalized by providing class instances for the existing Isabelle/HOL product type 'a × 'b. Isabelle type classes with overloaded constants are instantiated with the **instantiation** command (for classes that fix no new constants, a simpler **instance** command suffices). With the following **instantiation** command block, we define the ordering on type 'a × 'b (which is infix syntax for the type ('a, 'b) prod) in terms of the orderings on types 'a and 'b, which are in turn assumed to be in class below. Because class below asserts no axioms, the instance proof is trivial (using the ".." proof method).

```
instantiation prod :: (below, below) below
begin
  definition below_prod_def:
    "(op ⊑) ≡ (λp1 p2. fst p1 ⊑ fst p2 ∧ snd p1 ⊑ snd p2)"
  instance ..
end
```

Next, the following instance declarations assert that type 'a × 'b is in class po whenever both 'a and 'b are, and that a similar relationship holds for classes cpo and pcpo. Each **instance** command yields a proof obligation, but the proofs are completely standard, so we omit them here.

```
instance prod :: (po, po) po
instance prod :: (cpo, cpo) cpo
instance prod :: (pcpo, pcpo) pcpo
```

We prove all the usual theorems about ordering, least upper bounds, continuity, and strictness for the operations of Pair, fst, and snd.

lemma Pair_below_iff: "(a, b) ⊑ (c, d) ⟷ a ⊑ c ∧ b ⊑ d"

lemma lub_Pair: "⟦chain X; chain Y⟧ ⟹ (⨆i. (X i, Y i)) = (⨆i. X i, ⨆i. Y i)"

lemma cont_fst: "cont fst"

lemma cont_snd: "cont snd"

lemma cont2cont_Pair: "⟦cont f; cont g⟧ ⟹ cont (λx. (f x, g x))"

lemma fst_strict: "fst ⊥ = ⊥"

lemma snd_strict: "snd ⊥ = ⊥"

lemma Pair_strict: "(⊥, ⊥) = ⊥"

One lemma that is new in HOLCF '11 is the compactness property for the Pair constructor:

lemma compact_Pair: "⟦compact x; compact y⟧ ⟹ compact (x, y)"

The proof starts by applying the standard introduction rule for compactness, so we have the goal adm (λz. (x, y) ⋢ z). Simplifying with the definition of ordering on the product type, this becomes adm (λz. x ⋢ fst z ∨ y ⋢ snd z). Finally, this is solved using lemmas adm_disj, adm_compact_not_below, cont_fst, and cont_snd, with the assumptions about x and y.

2.4.2 Full function space cpo

If D is an arbitrary set, and E is a cpo, then the full function space $D \Rightarrow E$ forms a cpo when the functions are given the pointwise ordering. If E has a bottom element, then $D \Rightarrow E$ contains a minimal element.

HOLCF '11 provides the following class instances for the full function space type 'a ⇒ 'b. As with the product type, the proofs are the same standard ones used in previous HOLCF versions, so we just summarize the basic results.

instance fun :: (type, po) po

instance fun :: (type, cpo) cpo

```
instance fun :: (type, pcpo) pcpo
```

We prove the usual lemmas about ordering, least upper bounds, strictness and continuity:

```
lemma fun_below_iff: "(f ⊑ g) = (∀x. f x ⊑ g x)"
lemma thelub_fun: "chain (Y::nat ⇒ 'a ⇒ 'b) ⟹ (⨆i. Y i) = (λx. ⨆i. Y i x)"
lemma app_strict: "⊥ x = ⊥"
lemma cont2cont_fun: "cont f ⟹ cont (λx. f x y)"
lemma cont2cont_lambda: "⟦⋀y. cont (λx. f x y)⟧ ⟹ cont (λx y. f x y)"
```

2.4.3 Continuous function type

In domain theory, the continuous function space $[D \to E]$ consists of the set of all continuous functions between cpos D and E. Ordered pointwise, $[D \to E]$ forms a cpo, and it has a least element whenever E does. In HOLCF '11, we define the continuous function type 'a → 'b to be isomorphic to a subset of the full function type 'a ⇒ 'b, using the **cpodef** command.

```
cpodef ('a, 'b) cfun (infixr "→" 0) = "{f::('a::cpo ⇒ 'b::cpo). cont f}"
```

Two proof obligations are required for this definition to be accepted: For nonemptiness, we must show that there exists a continuous function of type 'a ⇒ 'b (any constant function will do). For admissibility, we must show that the lub of a chain of continuous functions is continuous, which is an easy corollary of lemma cont2cont_lub (Sec. 2.2.2). Once the type definition is processed we get the following representation and abstraction functions.

```
Rep_cfun :: ('a → 'b) ⇒ ('a ⇒ 'b)
Abs_cfun :: ('a ⇒ 'b) ⇒ ('a → 'b)
```

The HOLCF notations for continuous application and abstraction are really just abbreviations for these constants: The syntax f·x represents Rep_cfun f x, and

```
definition cfun_def: "cfun ≡ {f. cont f}"
theorem Rep_cfun_inverse: "Abs_cfun (Rep_cfun x) = x"
theorem Abs_cfun_inverse: "y ∈ cfun ⟹ Rep_cfun (Abs_cfun y) = y"
theorem Rep_cfun: "Rep_cfun x ∈ cfun"
theorem Rep_cfun_inject: "(Rep_cfun x = Rep_cfun y) = (x = y)"
definition below_cfun_def: "(op ⊑) ≡ (λx y. Rep_cfun x ⊑ Rep_cfun y)"
theorem cont_Rep_cfun: "cont f ⟹ cont (λx. Rep_cfun (f x))"
theorem cont_Abs_cfun: "⟦⋀x. f x ∈ cfun; cont f⟧ ⟹ cont (λx. Abs_cfun (f x))"
```

Figure 2.7: Selected theorems generated by **cpodef** for continuous function type

(Λ x. t) represents Abs_cfun (λx. t).

The **cpodef** command also generates several theorems, including those listed in Fig. 2.7. Some of these are produced by the Typedef package, but the last few are added by Cpodef.

Several of the key properties about the continuous function space are derived fairly directly from the theorems produced by Cpodef; Fig. 2.8 lists these. Extensionality follows from Rep_cfun_inject; extensionality of function inequality is a consequence of below_cfun_def. We get eta-conversion from Rep_cfun_inverse, while beta-conversion (with a continuity side-condition) follows from Abs_cfun_inverse. The continuous application operator Rep_cfun can be proved continuous in its first argument using cont_Rep_cfun, and in its second using Rep_cfun; these lemmas can then be combined into a single continuity rule using lemma cont_apply. Finally, a continuity rule for continuous abstraction follows from rule cont_Abs_cfun.

The **cpodef** command automatically proves that type 'a → 'b is an instance of class cpo (as long as 'a and 'b are also cpos, as required by the definition). But we should also like to reason about the bottom element of type 'a → 'b, which must exist whenever type 'b is pointed. Therefore we prove the following class instance.

```
lemma cfun_eq_iff: "(f = g) = (∀x. f·x = g·x)"

lemma cfun_below_iff: "(f ⊑ g) = (∀x. f·x ⊑ g·x)"

lemma eta_cfun: "(Λ x. f·x) = f"

lemma beta_cfun: "cont f ⟹ (Λ x. f x)·u = f u"

lemma cont_Rep_cfun1: "cont (λf. f·x)"

lemma cont_Rep_cfun2: "cont (λx. f·x)"

lemma cont2cont_APP: "⟦cont (λx. f x); cont (λx. t x)⟧ ⟹ cont (λx. (f x)·(t x))"

lemma cont2cont_LAM:
  "⟦⋀x. cont (λy. f x y); ⋀y. cont (λx. f x y)⟧ ⟹ cont (λx. Λ y. f x y)"
```

Figure 2.8: Properties of continuous functions, derived from **cpodef** theorems

```
instance cfun :: (cpo, pcpo) pcpo
```

We prove the instance using typedef_pcpo, the same lemma used internally by the **pcpodef** command. As a proof obligation, we must show that (⊥::'a ⇒ 'b) ∈ cfun. From the theory of the full function space, we know that ⊥ = (λx. ⊥), which is continuous because it is a constant function. Using lemmas typedef_Rep_strict and typedef_Abs_strict, we can then show that Rep_cfun and Abs_cfun are strict:

```
lemma APP_strict: "⊥·x = ⊥"

lemma LAM_strict: "(Λ x. ⊥) = ⊥"
```

HOLCF '11 also defines a few useful operations for the continuous function type. The identity and composition operator are straightforward and require little discussion; they have been present in all versions of HOLCF. The abstractions used in their definitions are easily shown to be continuous, so there is no obstacle to unfolding and beta-reducing their definitions. The theory defines infix syntax f oo g to represent cfcomp·f·g.

```
definition ID :: "'a → 'a"
  where "ID = (Λ x. x)"
```

```
definition cfcomp :: "('b → 'c) → ('a → 'b) → 'a → 'c"
  where "cfcomp = (Λ f g x. f·(g·x))"
```

The next operation in the continuous function theory is seq, which can be used to construct strict functions; it is new for HOLCF '11. It takes two arguments x and y; the result is $\bot$ when $x = \bot$ and y otherwise.[4] In the upcoming sections, we will see that it is useful for defining several operations on the strict product and strict sum types.

```
definition seq :: "'a::pcpo → 'b::pcpo → 'b"
  where "seq = (Λ x. if x = ⊥ then ⊥ else ID)"
```

Proving that the body of seq is continuous is a bit more difficult than for ID or cfcomp. To prove continuity, we must show for an arbitrary chain Y that $(\text{if } (\bigsqcup i.\ Y\ i) = \bot \text{ then } \bot \text{ else ID})$ is a least upper bound of the sequence given by $(\lambda i.\ \text{if } Y\ i = \bot \text{ then } \bot \text{ else ID})$. The proof proceeds using a lemma about least upper bounds:

```
lemma lub_eq_bottom_iff: "chain Y ⟹ (⨆i. Y i) = ⊥ ⟷ (∀i. Y i = ⊥)"
```

By unfolding the definition of least upper bound, and using lub_eq_bottom_iff as a rewrite rule, we can prove continuity of seq using Isabelle's simplifier. The simplifier automatically performs the necessary case splits on if-then-else expressions. We prove the following rules about seq, and add some of them to the simplifier:

```
lemma seq_conv_if: "seq·x = (if x = ⊥ then ⊥ else ID)"

lemma seq_simps [simp]:
  "seq·⊥ = ⊥"
  "seq·x·⊥ = ⊥"
  "x ≠ ⊥ ⟹ seq·x = ID"
```

The older HOLCF function strictify, which takes a function argument and returns a strict function, is now defined in terms of seq as follows.

[4] Haskell programmers may recognize this as the Haskell primitive `seq`, which has the same semantics.

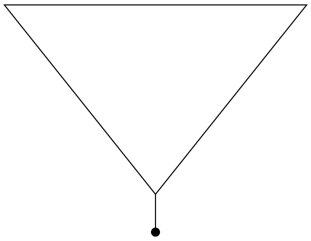


Figure 2.9: Lifted cpo

```
definition strictify :: "('a::pcpo → 'b::pcpo) → 'a → 'b"
  where "strictify = (Λ f x. seq·x·(f·x))"
```

2.4.4 Lifted cpo

Lifting is a way to create a new pointed cpo from any given cpo, by adding a new bottom element (see Fig. 2.9). If D is a cpo (which may or may not have a least element), then the lifted cpo $D_\bot$ consists of $\bot$ and values of the form $up(x)$ where $x \in D$. Furthermore, the constructor $up : D \to D_\bot$ is non-strict, that is, $up(x) \neq \bot$ for any $x \in D$. In particular, if D has a bottom element we have $up(\bot) \neq \bot$.

In HOLCF, the lifted cpo is formalized by the type constructor 'a u, which is defined using the Isabelle/HOL DATATYPE package.[5] (We also define $'a_\bot$ as an alternative syntax to 'a u.)

```
datatype 'a u = Ibottom | Iup 'a
```

Next we instantiate class below for type $'a_\bot$ (requiring 'a to be in class cpo). We define the ordering in the expected way, with Ibottom as the least element.

```
instantiation u :: (cpo) below
begin
  definition below_up_def: "(x ⊑ y) =
    (case x of Ibottom ⇒ True | Iup a ⇒
```

[5] The name 'a u was inherited from earlier LCF provers. Perhaps 'a lift would be a more apt name, but it is already taken in HOLCF (see §2.4.5).

```
      (case y of Ibottom ⇒ False | Iup b ⇒ a ⊑ b))"
  instance ..
end
```

It is straightforward to show that type $'a_{\bot}$ is a partial order: The proofs of reflexivity, transitivity, and antisymmetry proceed by case analysis. Proving the cpo instance requires more work. In order to show chain-completeness of the lifted cpo type, we prove a case analysis lemma for chains of type nat ⇒ $'a_{\bot}$.

```
lemma up_chain_lemma:
  assumes "chain Y"
  shows "(∀i. Y i = Ibottom) ∨
    (∃A k. (∀i. Iup (A i) = Y (i + k)) ∧ chain A ∧ range Y <<| Iup (⨆i. A i))"
```

In the first case, a chain Y might be constantly Ibottom: In this case the least upper bound of Y is Ibottom. Otherwise, Y has an initial segment of zero or more Ibottom terms, followed by the Iup constructor mapped over a chain A :: nat ⇒ 'a. In this case, Iup (⨆ i. A i) gives the least upper bound of chain Y; showing this requires a lemma stating that shifting a sequence preserves the least upper bound. Thus, by case analysis on chains, we prove that $'a_{\bot}$ is a cpo. Proving the pcpo class instance is easy, because Ibottom is the minimal element of the type.

With the class instances finished, we can now define operations on type $'a_{\bot}$ with continuous function types. We start with the constructor up.

```
definition up :: "'a → 'a⊥" where "up = (Λ a. Iup a)"
```

It is relatively simple to prove that Iup preserves least upper bounds (and is therefore continuous), by unfolding the definitions and calling the simplifier. Now knowing Ibottom = ⊥ and Iup x = up·x, we can prove new versions of the case-analysis and induction rules for type $'a_{\bot}$ in terms of ⊥ and up instead of Ibottom and Iup. We similarly derive the ordering properties and injectivity of up from the same properties of Iup.

```
lemma up_defined [simp]: "up·x ≠ ⊥"
lemma up_eq [simp]: "(up·x = up·y) = (x = y)"
lemma up_below [simp]: "(up·x ⊑ up·y) = (x ⊑ y)"
```

To prove that the up constructor preserves compactness requires unfolding the definitions and reasoning by cases on chains, using up_chain_lemma.

```
lemma compact_up: "compact x ⟹ compact (up·x)"
```

Next, we define the case combinator for type $\text{'a}_\perp$, called fup.

```
definition fup :: "('a → 'b::pcpo) → 'a⊥ → 'b"
  where "fup = (Λ f x. case x of Ibottom ⇒ ⊥ | Iup a ⇒ f·a)"
```

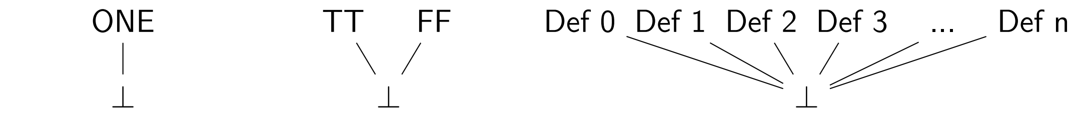
It is easy to prove that the body of fup is continuous in f, by case analysis on x. To show that the body of fup is continuous in x requires using up_chain_lemma again, along with more reasoning about least upper bounds of shifted chains. The case combinator obeys the following rules:

```
lemma fup_simps [simp]:
  "fup·f·⊥ = ⊥"
  "fup·f·(up·x) = f·x"
```

We also define special syntax for fup. The case expression (case x of up·a ⇒ t) represents the term fup·(Λ a. t)·x, and (Λ(up·a). t) represents fup·(Λ a. t).

2.4.5 Cpos from HOL types

In this section we will introduce two type constructors that facilitate the integration of types and terms from Isabelle/HOL with HOLCF [MNOS99]. The first, 'a discr (introduced in HOLCF '99), yields an unpointed discrete cpo. The second, 'a lift (present since HOLCF '95), adds a bottom element to yield a flat pointed cpo. The 'a lift type constructor is also used to model two of the base types from LCF: the lifted unit type one, and the domain of truth values tr (see Fig. 2.10).



(a) one = unit lift (b) tr = bool lift (c) nat lift

Figure 2.10: Flat lifted types in HOLCF

Discrete cpo. Any ordinary HOL type can be turned into a cpo by giving it a discrete ordering. In HOLCF this construction is formalized using the type 'a discr.

```
datatype 'a discr = Discr "'a"
```

To go along with the constructor function Discr, we define its inverse, undiscr.

```
definition undiscr :: "'a discr ⇒ 'a"
  where "undiscr x = (case x of Discr y ⇒ y)"
```

The ordering on type 'a discr is defined so that $(x \sqsubseteq y) = (x = y)$, which makes 'a discr an instance of the discrete_cpo class (and thus also the chfin class). In particular, this means that every function f :: 'a discr ⇒ 'b is continuous, and every predicate P :: 'a discr ⇒ bool is admissible.

It might make sense to define discrete orderings for various HOL types, such as bool, nat, and int, making all those types instances of the discrete_cpo class. However, this is not really necessary, because the types bool discr, nat discr, and int discr can be used instead.

HOLCF '11 does define a discrete ordering for the single-element HOL unit type. Type unit is special, because it is the only discrete cpo that also has a bottom element. For technical reasons, it is much simpler to make unit an instance of class pcpo directly than it would be to make unit discr :: pcpo. The Isabelle type unit corresponds to the void type from Cambridge LCF [Pau87].

Flat lifted type. Any ordinary HOL type can be made into a pointed cpo by adjoining a new bottom element, and giving the new type a flat ordering (in the sense of the flat type class described in Sec. 2.2.1). This construction is formalized in HOLCF by the type 'a lift.

```
pcpodef (open) 'a lift = "UNIV :: ('a discr)⊥ set"
```

We use the Cpodef package to define 'a lift as an isomorphic copy of ('a discr)$_\perp$.[6,7] As such, the Rep_lift and Abs_lift functions form a continuous isomorphism. Next we define the constructor function, Def.

```
definition Def :: "'a ⇒ 'a lift"
  where "Def x = Abs_lift (up·(Discr x))"
```

Using this definition, we prove that Def is injective and never returns ⊥. It is also simple to prove an induction rule for type 'a lift, by case analysis on the representing type.

```
lemma lift_induct: "⟦P ⊥; ⋀x. P (Def x)⟧ ⟹ P y"
```

Using these lemmas, we then use the rep_datatype command provided by the Datatype package to generate a case combinator lift_case and various related theorems; thus we are allowed to treat Def and ⊥ as constructors of an ordinary datatype.

By case analysis, and using the ordering properties of Def, we can show that 'a lift is an instance of class flat. Because flat is a subclass of chfin, this means also that every value of type 'a lift is compact.

In order to use lift_case to define continuous functions, we need to prove a continuity rule for it. We start with a lemma that lets us express lift_case using Rep_lift and fup. Having expressed lift_case in terms of other continuous functions, it is now easy to show the continuity of lift_case.

```
lemma lift_case_eq:
  "(case x of ⊥ ⇒ ⊥ | Def a ⇒ f a) = (case Rep_lift x of up·y ⇒ f (undiscr y))"
```

[6] The (**open**) option suppresses the definition of the set constant lift ≡ UNIV.

[7] Another sensible design choice would be to use **type_synonym** 'a lift = "'a discr u".

```
lemma cont2cont_lift_case [simp]:
  "⟦⋀a. cont (λx. f x a); cont (λx. g x)⟧
    ⟹ cont (λx. case g x of ⊥ ⇒ ⊥ | Def a ⇒ f x a)"
```

To facilitate the definition of continuous functions on type 'a lift, we define combinators flift1 and flift2. From cont2cont_lift_case, we derive a continuity rule for flift1 (not shown).

```
definition flift1 :: "('a ⇒ 'b::pcpo) ⇒ ('a lift → 'b)"
  where "flift1 f = (Λ x. case x of ⊥ ⇒ ⊥ | Def a ⇒ f a)"

definition flift2 :: "('a ⇒ 'b) ⇒ ('a lift → 'b lift)"
  where "flift2 f = flift1 (λx. Def (f x))"
```

We also define the syntax (Λ(Def x). t) to represent flift1 (λx. t).

Lifted unit type. We define the standard LCF type one as a type synonym for unit lift. We proceed to define the constructor ONE and a case combinator one_case (strict in its second argument), as shown. We also prove the expected ordering properties and rewrite rules for these constants.

```
type_synonym one = "unit lift"

definition ONE :: "one"
  where "ONE = Def ()"

definition one_case :: "'a → one → 'a"
  where "one_case = (Λ x (Def u). x)"
```

We define the syntax (case x of ONE ⇒ t) to represent one_case·t·x, and (Λ ONE. t) to represent one_case·t.

Lifted boolean type. The LCF type of truth values, tr, is defined as a synonym for bool lift. This type has constructors TT and FF; its case combinator tr_case is the continuous if-then-else operator.

```
type_synonym tr = "bool lift"
```

```
definition TT :: "tr"
  where "TT = Def True"

definition FF :: "tr"
  where "FF = Def False"

definition tr_case :: "'a → 'a → tr → 'a"
  where "tr_case = (Λ x y (Def b). if b then x else y)"
```

We define the syntax (If t then x else y) to represent tr_case·x·y·t. Note that we use "If" with a capital letter to distinguish from the standard if-then-else operation on type bool, which uses lowercase "if".

2.4.6 Strict product type

The strict product (also known as "smash product") is one of the standard constructions in domain theory [GS90, Gun92, AC98]. If D and E are pointed cpos, then $D \otimes E$ is also a pointed cpo. Its elements are strict pairs of the form $(\!|x, y|\!)$, where $x \in D$ and $y \in E$. Strict pairs where either component equals $\bot$ are identified with the bottom element of $D \otimes E$, so that $(\!|x, \bot|\!) = (\!|\bot, y|\!) = \bot$.

In HOLCF '11, we define the strict product of 'a and 'b as a subset of the cartesian product type 'a × 'b. The type consists of those pairs where neither element is $\bot$, plus the pair $\bot = (\bot, \bot)$.

```
pcpodef ('a, 'b) sprod  (infixr "⊗" 20) =
  "{p::('a::pcpo × 'b::pcpo). p = ⊥ ∨ (fst p ≠ ⊥ ∧ snd p ≠ ⊥)}"
```

The proof obligations for **pcpodef** are that the set contains $\bot$, and set membership is admissible. Both goals are solved immediately by simplification. (Note that as $\bot$ is compact, being non-$\bot$ is an admissible predicate.) The CPODEF package generates a set of lemmas analogous to those listed in Fig. 2.7, plus a few more about the strictness of Rep_sprod and Abs_sprod (see Fig. 2.11).

```
definition sprod_def: "sprod ≡ {p. p = ⊥ ∨ (fst p ≠ ⊥ ∧ snd p ≠ ⊥)}"
theorem Rep_sprod_inverse: "Abs_sprod (Rep_sprod x) = x"
theorem Abs_sprod_inverse: "y ∈ sprod ⟹ Rep_sprod (Abs_sprod y) = y"
theorem Rep_sprod: "Rep_sprod x ∈ sprod"
theorem Rep_sprod_inject: "(Rep_sprod x = Rep_sprod y) = (x = y)"
definition below_sprod_def: "(op ⊑) ≡ (λx y. Rep_sprod x ⊑ Rep_sprod y)"
theorem cont_Rep_sprod: "cont f ⟹ cont (λx. Rep_sprod (f x))"
theorem cont_Abs_sprod: "⟦⋀x. f x ∈ sprod; cont f⟧ ⟹ cont (λx. Abs_sprod (f x))"
theorem compact_sprod: "compact (Rep_sprod x) ⟹ compact x"
theorem Rep_sprod_strict: "Rep_sprod ⊥ = ⊥"
theorem Abs_sprod_strict: "Abs_sprod ⊥ = ⊥"
```

Figure 2.11: Selected theorems generated by **pcpodef** for strict product type

Constructor function. The first operation we define is the strict pair constructor, spair. The definition uses the operator seq to ensure that if either a or b is ⊥, then both components of the resulting pair will be ⊥.

```
definition spair :: "'a → 'b → ('a ⊗ 'b)"
  where "spair = (Λ a b. Abs_sprod (seq·b·a, seq·a·b))"
```

We also define syntax for strict tuples. The expression (:a, b:) is syntax for spair·a·b; larger tuples nest to the right, so (:a, b, c:) means spair·a·(spair·b·c).

In order to prove properties about spair, we need to know how Rep_sprod acts on it. We start by proving (seq·b·a, seq·a·b) ∈ sprod as a lemma. Then we can use cont_Abs_sprod and Abs_sprod_inverse (provided by the Cpodef package) to beta-reduce the definition of spair and prove the desired rule:

```
lemma Rep_sprod_spair: "Rep_sprod (:a, b:) = (seq·b·a, seq·a·b)"
```

Together with a group of other rewrite rules, the lemma Rep_sprod_spair forms part of a general strategy for proving properties about strict products. The rules in Rep_sprod_simps replace comparisons on type 'a ⊗ 'b with comparisons on the component types 'a and 'b.

```
lemma Rep_sprod_simps:
  "x = y ⟷ Rep_sprod x = Rep_sprod y"
  "x ⊑ y ⟷ Rep_sprod x ⊑ Rep_sprod y"
  "t = u ⟷ fst t = fst u ∧ snd t = snd u"
  "t ⊑ u ⟷ fst t ⊑ fst u ∧ snd t ⊑ snd u"
  "Rep_sprod (:a, b:) = (seq·b·a, seq·a·b)"
  "Rep_sprod ⊥ = ⊥"
```

Each of the following five lemmas is proved automatically by simplifying with Rep_sprod_simps. (The if-and-only-if rules also require seq_conv_if, so that the simplifier will do the appropriate case splits on whether or not values equal ⊥.)

```
lemma spair_strict1 [simp]: "(:⊥, b:) = ⊥"

lemma spair_strict2 [simp]: "(:a, ⊥:) = ⊥"

lemma spair_bottom_iff [simp]: "(:a, b:) = ⊥ ⟷ a = ⊥ ∨ b = ⊥"

lemma spair_below_iff: "(:a, b:) ⊑ (:c, d:) ⟷
  a = ⊥ ∨ b = ⊥ ∨ (a ⊑ c ∧ b ⊑ d)"

lemma spair_eq_iff: "(:a, b:) = (:c, d:) ⟷
  (a = c ∧ b = d) ∨ ((a = ⊥ ∨ b = ⊥) ∧ (c = ⊥ ∨ d = ⊥))"
```

The same set of rewrite rules also works for proving a case analysis rule for type 'a ⊗ 'b. The proof of sprodE starts by inserting the fact Rep_sprod y ∈ sprod into the goal state. Then we unfold the definition of sprod, and use the auto tactic to split the disjunctions and simplify.

```
lemma sprodE:
  "⟦y = ⊥ ⟹ P; ⋀a b. ⟦y = (:a, b:); a ≠ ⊥; b ≠ ⊥⟧ ⟹ P⟧ ⟹ P"
```

The induction rule for strict products is a direct consequence of the case analysis rule.

```
lemma sprod_induct: "⟦P ⊥; ⋀a b. ⟦a ≠ ⊥; b ≠ ⊥⟧ ⟹ P (:a, b:)⟧ ⟹ P x"
```

One more useful rule about spair concerns compactness. We can use the lemma compact_sprod provided by the CPODEF package together with Rep_sprod_spair to prove the following rule.

```
lemma compact_spair: "⟦compact a; compact b⟧ ⟹ compact (:a, b:)"
```

Projections. After the constructor spair, the next functions we define are the projections sfst and ssnd. We can easily unfold and beta-reduce their definitions, because fst, snd, and Rep_sprod are all known to be continuous.

```
definition sfst :: "('a ⊗ 'b) → 'a"
  where "sfst = (Λ p. fst (Rep_sprod p))"

definition ssnd :: "('a ⊗ 'b) → 'b"
  where "ssnd = (Λ p. snd (Rep_sprod p))"
```

The basic properties of sfst and ssnd can be proven by beta-reducing their definitions, and simplifying with Rep_sprod_simps.

```
lemma sfst_strict [simp]: "sfst·⊥ = ⊥"

lemma ssnd_strict [simp]: "ssnd·⊥ = ⊥"

lemma sfst_spair [simp]: "y ≠ ⊥ ⟹ sfst·(:x, y:) = x"

lemma ssnd_spair [simp]: "x ≠ ⊥ ⟹ ssnd·(:x, y:) = y"
```

Case combinator. The last operation on strict products that we need to define is the case combinator, ssplit.

```
definition ssplit :: "('a → 'b → 'c) → ('a ⊗ 'b) → 'c"
  where "ssplit = (Λ f p. seq·p·(f·(sfst·p)·(ssnd·p)))"
```

The characteristic lemmas for ssplit are proved easily by unfolding the definition. Note that the use of seq in the definition of ssplit is needed for the strictness rule to hold.

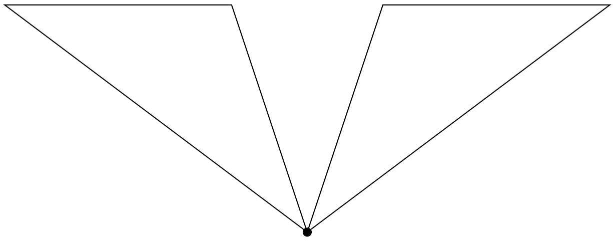


Figure 2.12: Strict sum of pointed cpos

```
lemma ssplit_simps [simp]:
  "ssplit·f·⊥ = ⊥"
  "⟦x ≠ ⊥; y ≠ ⊥⟧ ⟹ ssplit·f·(:x, y:) = f·x·y"
```

The syntax (case x of (:a, b:) ⇒ t) stands for ssplit·(Λ a b. t)·x. We also support lambda syntax: (Λ (:a, b:). t) is shorthand for ssplit·(Λ a b. t). Larger tuples work too; for example, (Λ (:a, b, c:). t) translates to ssplit·(Λ a. ssplit·(Λ b c. t)).

2.4.7 Strict sum type

The strict sum $D \oplus E$ (also known as "smash sum" or "coalesced sum") is another standard construction in domain theory [GS90, Gun92, AC98]. If D and E are pointed cpos, then $D \oplus E$ is also a pointed cpo. There are continuous injections $\iota_1 : D \to D \oplus E$ and $\iota_2 : E \to D \oplus E$, similar to the disjoint sum. But unlike the disjoint sum, we identify the bottom elements injected from D and E, making each injection strict: $\iota_1(\bot) = \iota_2(\bot) = \bot$ (see Fig. 2.12).

In HOLCF '11, we define the strict sum of 'a and 'b as a set of triples of type (tr × 'a × 'b). The first component is a tag specifying whether the value should be interpreted as a left- or right-injection. Depending on the value of the tag, the injected value is stored in either the second or third slot, with the other slot containing ⊥. The bottom element of the strict sum type is represented as $\bot = (\bot, \bot, \bot)$.

```
pcpodef ('a, 'b) ssum (infixr "⊕" 10) =
  "{p :: (tr × 'a::pcpo × 'b::pcpo). p = ⊥ ∨
```

```
      (fst p = TT ∧ fst (snd p) ≠ ⊥ ∧ snd (snd p) = ⊥) ∨
      (fst p = FF ∧ fst (snd p) = ⊥ ∧ snd (snd p) ≠ ⊥) }"
```

The proof obligations for **pcpodef** are that the set contains ⊥, and set membership is admissible. As with the strict product, both goals are solved immediately by simplification. For the strict sum type, the CPODEF package provides a set of theorems just like the ones shown in Fig. 2.11 for strict product; we omit the strict sum versions here.

Constructor functions. After defining the type, we can define the constructor functions sinl and sinr.[8] Because both constructor functions are supposed to be strict, we use the seq operator to ensure that the tag component is ⊥ whenever the constructor argument is ⊥.

```
definition sinl :: "'a → ('a ⊕ 'b)"
  where "sinl = (Λ a. Abs_ssum (seq·a·TT, a, ⊥))"

definition sinr :: "'b → ('a ⊕ 'b)"
  where "sinr = (Λ b. Abs_ssum (seq·b·FF, ⊥, b))"
```

Similarly to the strict product, the next thing we must do is prove how Rep_ssum acts on the constructors. After proving lemmas to the effect that (seq·a·TT, a, ⊥) and (seq·b·FF, ⊥, b) are always in the set ssum, we can use cont_Abs_ssum and Abs_ssum_inverse to derive the following two rules.

```
lemma Rep_ssum_sinl: "Rep_ssum (sinl·a) = (seq·a·TT, a, ⊥)"

lemma Rep_ssum_sinr: "Rep_ssum (sinr·b) = (seq·b·FF, ⊥, b)"
```

These are combined with other lemmas about Rep_ssum to form Rep_ssum_simps, which is the strict-sum analog of Rep_sprod_simps from the previous section. This set of rewrite rules lets us reduce comparisons on type 'a ⊕ 'b to comparisons on

[8] The names of sinl and sinr come from earlier versions of LCF; respectively they stand for “*s*trict-*in*jection-*l*eft” and “*s*trict-*in*jection-*r*ight”.

lemma sinl_below [simp]: "$(\mathsf{sinl}\cdot\mathsf{x} \sqsubseteq \mathsf{sinl}\cdot\mathsf{y}) = (\mathsf{x} \sqsubseteq \mathsf{y})$"

lemma sinr_below [simp]: "$(\mathsf{sinr}\cdot\mathsf{x} \sqsubseteq \mathsf{sinr}\cdot\mathsf{y}) = (\mathsf{x} \sqsubseteq \mathsf{y})$"

lemma sinl_below_sinr [simp]: "$(\mathsf{sinl}\cdot\mathsf{x} \sqsubseteq \mathsf{sinr}\cdot\mathsf{y}) = (\mathsf{x} = \bot)$"

lemma sinr_below_sinl [simp]: "$(\mathsf{sinr}\cdot\mathsf{x} \sqsubseteq \mathsf{sinl}\cdot\mathsf{y}) = (\mathsf{x} = \bot)$"

lemma sinl_eq [simp]: "$(\mathsf{sinl}\cdot\mathsf{x} = \mathsf{sinl}\cdot\mathsf{y}) = (\mathsf{x} = \mathsf{y})$"

lemma sinr_eq [simp]: "$(\mathsf{sinr}\cdot\mathsf{x} = \mathsf{sinr}\cdot\mathsf{y}) = (\mathsf{x} = \mathsf{y})$"

lemma sinl_eq_sinr [simp]: "$(\mathsf{sinl}\cdot\mathsf{x} = \mathsf{sinr}\cdot\mathsf{y}) = (\mathsf{x} = \bot \wedge \mathsf{y} = \bot)$"

lemma sinr_eq_sinl [simp]: "$(\mathsf{sinr}\cdot\mathsf{x} = \mathsf{sinl}\cdot\mathsf{y}) = (\mathsf{x} = \bot \wedge \mathsf{y} = \bot)$"

lemma sinl_strict [simp]: "$\mathsf{sinl}\cdot\bot = \bot$"

lemma sinr_strict [simp]: "$\mathsf{sinr}\cdot\bot = \bot$"

lemma sinl_bottom_iff [simp]: "$(\mathsf{sinl}\cdot\mathsf{x} = \bot) = (\mathsf{x} = \bot)$"

lemma sinr_bottom_iff [simp]: "$(\mathsf{sinr}\cdot\mathsf{x} = \bot) = (\mathsf{x} = \bot)$"

Figure 2.13: Order, injectivity, distinctness, and strictness of sinl and sinr

types tr, 'a and 'b. We can use them (together with seq_conv_if, as needed) to prove the complete set of simplification rules for sinl and sinr shown in Fig. 2.13.

The same rewrites are also sufficient to prove the case analysis rule for the strict sum type. Starting with the fact Rep_ssum y ∈ ssum (a theorem provided by Cpodef), we can unfold the definition of ssum and then use Rep_ssum_simps with the auto tactic to complete the proof.

lemma ssumE: "$[\![\mathsf{y} = \bot \Longrightarrow \mathsf{P};$
$\bigwedge\mathsf{a}.\ [\![\mathsf{y} = \mathsf{sinl}\cdot\mathsf{a};\ \mathsf{a} \neq \bot]\!] \Longrightarrow \mathsf{P};\ \bigwedge\mathsf{b}.\ [\![\mathsf{y} = \mathsf{sinr}\cdot\mathsf{b};\ \mathsf{b} \neq \bot]\!] \Longrightarrow \mathsf{P}]\!] \Longrightarrow \mathsf{P}$"

The induction rule for strict sums is a direct consequence of the case analysis rule.

lemma ssum_induct:
"$[\![\mathsf{P}\ \bot;\ \bigwedge\mathsf{a}.\ \mathsf{a} \neq \bot \Longrightarrow \mathsf{P}\ (\mathsf{sinl}\cdot\mathsf{a});\ \bigwedge\mathsf{b}.\ \mathsf{b} \neq \bot \Longrightarrow \mathsf{P}\ (\mathsf{sinr}\cdot\mathsf{b})]\!] \Longrightarrow \mathsf{P}\ \mathsf{x}$"

We can prove if-and-only-if compactness rules for sinl and sinr. For the (⟸) direction, we use a compactness rule provided by Cpodef: A value of type 'a ⊕ 'b is compact if its representation in type tr × 'a × 'b is compact. The proof of the (⟹) direction uses the definition of compactness and the adm_subst rule. We assume compact (sinl·a), which means adm (λx. sinl·a ⋢ x); this implies adm (λx. sinl·a ⋢ sinl·x), which simplifies to adm (λx. a ⋢ x), which is the definition of compact a. The proof for sinr is identical.

```
lemma compact_sinl_iff [simp]: "compact (sinl·a) = compact a"
lemma compact_sinr_iff [simp]: "compact (sinr·b) = compact b"
```

Case combinator. Besides the constructors sinl and sinr, the other basic operation on strict sums is the case combinator, sscase. The definition uses Rep_ssum to examine its argument, and the continuous if-then-else operation on type tr to select which branch to take.

```
definition sscase :: "('a → 'c) → ('b → 'c) → ('a ⊕ 'b) → 'c"
  where "sscase = (Λ f g s. (λ(t, x, y). If t then f·x else g·y) (Rep_ssum s))"
```

Using lemma cont_Rep_ssum from the Cpodef package, it is easy to show that the abstractions used in the definition of sscase are continuous. Proving the characteristic properties of sscase is a simple matter of beta-reducing the definition and simplifying with Rep_ssum_simps.

```
lemma sscase_simps [simp]:
  "sscase·f·g·⊥ = ⊥"
  "x ≠ ⊥ ⟹ sscase·f·g·(sinl·x) = f·x"
  "y ≠ ⊥ ⟹ sscase·f·g·(sinr·y) = g·y"
```

As with the other case combinators, we define special syntax for sscase. The term sscase·(Λ a. t)·(Λ b. u)·x is rendered as (case x of sinl·a ⇒ t | sinr·b ⇒ u).

2.5 AUTOMATING CONTINUITY PROOFS

When reasoning in HOLCF, continuity goals pop up all the time: Every beta-reduction of a continuous function abstraction requires a continuity proof. Continuity proofs are also needed, for example, to show admissibility of predicates like $(\lambda x.\ f\ x \sqsubseteq g\ x)$. Because they occur so often in practice, it is important that continuity proofs be both automatic and efficient—they should involve a minimal number of proof steps, and be processed in a short amount of time.

In this section we discuss three alternative proof techniques for continuity, starting with the method used by previous versions of HOLCF. Following this, we introduce two novel proof methods for continuity, each of which offers improved efficiency in terms of number of proof steps and running time.

2.5.1 Original HOLCF continuity tactic

HOLCF '95 introduced a continuity tactic that repeatedly applied the following four continuity lemmas:

```
lemma cont_id: "cont (λx. x)"

lemma cont_const: "cont (λx. c)"

lemma cont2cont_APP:
  "⟦cont (λx. f x); cont (λx. t x)⟧ ⟹ cont (λx. (f x)·(t x))"

lemma cont2cont_LAM:
  "⟦⋀x. cont (λy. f x y); ⋀y. cont (λx. f x y)⟧ ⟹ cont (λx. Λ y. f x y)"
```

If we add these four rules to the simplifier, then we will be able to prove continuity automatically for any term written in the LCF sub-language—i.e., any term consisting of continuous applications, continuous abstractions, and variables and constants with continuous function types. Looking at the forms of the rules, we can see that the terms in the assumptions are all strictly smaller than the terms

```
lemma "cont (λx. Λ a b c. f·a·b·c·x)"
  apply (intro cont2cont_LAM)

 goal (8 subgoals):
 1. ⋀x a b. cont (λc. f·a·b·c·x)
 2. ⋀x a c. cont (λb. f·a·b·c·x)
 3. ⋀x b a. cont (λc. f·a·b·c·x)
 4. ⋀x b c. cont (λa. f·a·b·c·x)
 5. ⋀a x b. cont (λc. f·a·b·c·x)
 6. ⋀a x c. cont (λb. f·a·b·c·x)
 7. ⋀a b x. cont (λc. f·a·b·c·x)
 8. ⋀a b c. cont (λx. f·a·b·c·x)
```

Figure 2.14: Exponential blow-up using rule cont2cont_LAM

in the conclusions, showing that the process of applying rules must eventually terminate.

But while the proofs may be automatic, they are not efficient. The problem lies specifically with the rule cont2cont_LAM. Consider what happens when we have a term with multiple nested function abstractions (see Fig. 2.14). After repeatedly applying rule cont2cont_LAM as many times as possible, we are left with *eight* subgoals. In fact, the number of subgoals is equal to 2^n, where n is the number of nested lambdas.

However, it is immediately evident that many of these subgoals are really the same; there are only $(n+1)$ distinct subgoals, each requiring continuity with respect to a different variable. This suggests the possibility of solving continuity goals in less than an exponential number of steps.

2.5.2 Bottom-up continuity proofs

One possible strategy is to work bottom-up: First prove continuity for the small subterms, and then combine those proofs to get continuity over the larger terms.

```
have a0: "cont (λa. a)" by (rule cont_id)
from cont_const a0 have a1: "cont (λa. f·a)" by (rule cont2cont_APP)
from a1 cont_const have a2: "⋀b. cont (λa. f·a·b)" by (rule cont2cont_APP)

...

from ... have a4: "⋀b c x. cont (λa. f·a·b·c·x)" by (rule cont2cont_APP)
from ... have b4: "⋀a c x. cont (λb. f·a·b·c·x)" by (rule cont2cont_APP)
from ... have c4: "⋀a b x. cont (λc. f·a·b·c·x)" by (rule cont2cont_APP)
from ... have x4: "⋀a b c. cont (λx. f·a·b·c·x)" by (rule cont2cont_APP)

from c4 and a4 have a5: "⋀b x. cont (λa. Λ c. f·a·b·c·x)" by (rule cont2cont_LAM)
from c4 and b4 have b5: "⋀a x. cont (λb. Λ c. f·a·b·c·x)" by (rule cont2cont_LAM)
from c4 and x4 have x5: "⋀a b. cont (λx. Λ c. f·a·b·c·x)" by (rule cont2cont_LAM)

from b5 and a5 have a6: "⋀x. cont (λa. Λ b c. f·a·b·c·x)" by (rule cont2cont_LAM)
from b5 and x5 have x6: "⋀a. cont (λx. Λ b c. f·a·b·c·x)" by (rule cont2cont_LAM)

from a6 and x6 have x7: "cont (λx. Λ a b c. f·a·b·c·x)" by (rule cont2cont_LAM)
```

Figure 2.15: Bottom-up algorithm for proving continuity, using forward proof

Note that this is basically opposite to the introduction-rules approach, which works top-down. For each subterm, we construct a list of continuity theorems, one for each bound variable that occurs free in the subterm. For example, when proving cont (λx. Λ a b c. f·a·b·c·x), the subterm (Λ c. f·a·b·c·x) will have continuity theorems for the variables a, b and x (shown in Fig. 2.15 as a5, b5, and x5). We treat f like a constant, because it is free in the original goal.

Next we move up to the next larger subterm, (Λ b c. f·a·b·c·x), which gets continuity theorems for the variables a and x (shown as a6 and x6). Each of the new continuity theorems is derived from two earlier ones using the rule cont2cont_LAM. Note that rule b5 is used more than once; such re-use prevents the exponential blow-up that we had before.

The bottom-up algorithm is now implemented in HOLCF '11 (HOLCF/Tools/cont_proc.ML). It has the advantage of being fast: It solves continuity subgoals using a small number of rule applications (quadratic in the number of nested lambdas, rather than exponential). Another advantage is that without any extra work, the algorithm can return a whole list of continuity theorems that it proved for subterms. This is useful for doing multiple beta-reductions, for example when proving something like (Λ a b c. f·a·b·c·x)·u·v·w = f·u·v·w·x. This technique is used successfully by the DOMAIN package for internal proofs, where large subgoals of this form arise from large datatype definitions (see Sec. 4.3).

Compared to using introduction rules, the main disadvantage of the bottom-up algorithm is that it is difficult to extend the system to handle new constants. For example, when Müller introduced the lift type to allow mixing HOL and LCF terms, he was able to extend the continuity tactic by simply adding a couple of new rules, such as the one below for lift_case [Mül98, §5.2.2] [MNOS99, §4.3.2].

```
lemma cont2cont_lift_case:
  "⟦⋀y. cont (λx. f x y); cont (λx. g x)⟧
    ⟹ cont (λx. case g x of ⊥ ⇒ ⊥ | Def y ⇒ f x y)"
```

Similarly, we have continuity rules for operations on the HOL product type:

```
lemma cont2cont_Pair:
  "⟦cont (λx. f x); cont (λx. g x)⟧ ⟹ cont (λx. (f x, g x))"

lemma cont2cont_prod_case:
  "⟦⋀a b. cont (λx. f x a b); ⋀x b. cont (λa. f x a b);
    ⋀x a. cont (λb. f x a b); cont (λx. g x)⟧
    ⟹ cont (λx. case g x of (a, b) ⇒ f x a b)"
```

An ideal continuity prover should allow users to add support for new constants by adding such rules. But as implemented, the bottom-up continuity algorithm is not so easily extensible. Properly supporting rules like cont2cont_prod_case is especially troublesome because, like cont2cont_LAM, it has multiple hypotheses for the same subterm, and would require similar handling to avoid an exponential blow-up.

Yet the rule has a different form than cont2cont_LAM. A single algorithm that could handle the full range of possibilities for such rules would have to be rather sophisticated.

2.5.3 Efficient continuity rules using products

As it turns out, it is possible to design an extensible set of continuity introduction rules that avoids the exponential blow-up inherent in the original HOLCF '95 continuity tactic. The key to designing these rules is a property of continuous functions $f : D \times E \to F$. It is a standard result in domain theory that a function on a product type is continuous if and only if it is continuous in each component separately [AJ94, Lemma 3.2.6]. This property is formalized in the HOLCF '11 theory of products as lemma prod_cont_iff.

```
lemma prod_cont_iff:
  "cont f ⟷ (∀y. cont (λx. f (x, y))) ∧ (∀x. cont (λy. f (x, y)))"
```

We can use this theorem to combine the two premises of the cont2cont_LAM rule into one. Compare the two versions of this rule:

```
lemma cont2cont_LAM:
  "⟦⋀x. cont (λy. f x y); ⋀y. cont (λx. f x y)⟧ ⟹ cont (λx. Λ y. f x y)"

lemma cont2cont_LAM':
  "cont (λp. f (fst p) (snd p)) ⟹ cont (λx. Λ y. f x y)"
```

We can evaluate the behavior of the new cont2cont_LAM' rule by considering the same continuity goal we used before in Fig. 2.14. After applying the continuity rule cont2cont_LAM' three times, we are now left with just a single subgoal (see Fig. 2.16). The remaining goal can be solved by applying standard continuity rules (cont_id, cont_const, and cont2cont_APP) together with continuity rules for fst and snd. The complete set of rules (which we call cont2cont rules) for proving continuity of LCF terms is shown in Fig. 2.17. All of them are safe to add to the Isabelle simplifier.

```
lemma "cont (λx. Λ a b c. f·a·b·c·x)"
  apply (intro cont2cont_LAM')

goal (1 subgoal):
 1. cont (λp. f·(snd (fst (fst p)))·(snd (fst p))·(snd p)·(fst (fst (fst p))))
```

Figure 2.16: Efficient behavior of continuity introduction rule cont2cont_LAM'

```
lemma cont_id [simp]: "cont (λx. x)"

lemma cont_const [simp]: "cont (λx. c)"

lemma cont2cont_fst [simp]: "cont (λx. f x) ⟹ cont (λx. fst (f x))"

lemma cont2cont_snd [simp]: "cont (λx. f x) ⟹ cont (λx. snd (f x))"

lemma cont2cont_APP [simp]:
  "⟦cont (λx. f x); cont (λx. t x)⟧ ⟹ cont (λx. (f x)·(t x))"

lemma cont2cont_LAM' [simp]:
  "cont (λp. f (fst p) (snd p)) ⟹ cont (λx. Λ y. f x y)"
```

Figure 2.17: Complete set of efficient cont2cont rules for LCF terms

Unlike rule cont2cont_LAM, each rule in Fig. 2.17 has only one continuity premise for each subterm. Proving continuity with this set of rules requires more than a linear number of steps, though, because as lambdas are eliminated, the subgoal grows to a larger size: fst and snd are introduced on every bound variable. But the size increase is not exponential; it is merely quadratic in the number of nested lambdas. So, like the bottom-up algorithm, the cont2cont rules can discharge a continuity goal with nested lambdas in a quadratic number of steps. In practice, the run-time efficiency of the cont2cont rules is about the same as for the bottom-up continuity tactic. (Either can discharge a continuity goal with twenty nested lambdas in a fraction of a second, and the total run-time for each seems to be $O(n^3)$.)

The cont2cont rules are easily extensible. We can add new rules to the simplifier for each new constant we want (such as cont2cont_lift_case) without any trouble. For constants like prod_case, whose continuity rules require continuity of multiple variables over the same subterm, we can use prod_cont_iff to derive an efficient version of the rule, just like we did for cont2cont_LAM. For example, here is the efficient cont2cont rule for prod_case:

```
lemma cont2cont_prod_case' [simp]:
  "⟦cont (λp. f (fst p) (fst (snd p)) (snd (snd p))); cont (λx. g x)⟧
    ⟹ cont (λx. case g x of (a, b) ⇒ f x a b)"
```

For organizing continuity introduction rules, HOLCF '11 provides a special cont2cont theorem attribute. Declaring a lemma with [cont2cont] dynamically adds it to a list of theorems, which is also referred to by the name cont2cont. (Typically each cont2cont rule should also be added to the simplifier.) Users can then do continuity proofs with the proof method "intro cont2cont".

Using "intro cont2cont" is a bit faster (by about a factor of two) than calling "simp" on a continuity goal, because it has a smaller, targeted set of rules, and avoids deep recursive calls to the simplifier. To take advantage of this speedup in the common case, HOLCF '11 defines a special simplification procedure (or "simproc") for the beta reduction rule:

lemma beta_cfun: "cont f $\Longrightarrow$ (Λ x. f x)$\cdot$u = f u"

In HOLCF '11, beta_cfun is not declared with the [simp] attribute. Instead, the simplifier is configured to call a certain simproc when it sees a term matching the pattern (Λ x. f x)$\cdot$u. When the simproc is run, it tries to prove the goal cont f using the proof method "intro cont2cont". If the subproof succeeds, the simproc returns the unconditional equation (Λ x. f x)$\cdot$u = f u; if the subproof fails, then the simproc returns the ordinary conditional rule, which will then cause the simplifier to attempt to solve the side-condition cont f by calling itself recursively.

A nice feature of this setup is that users do not have to know anything about the cont2cont attribute; if a user proves a continuity rule for a new constant, and declares it with [simp], then it will work with the other existing cont2cont rules as expected. Declaring the same rule with [cont2cont] is optional, and will not affect the set of provable continuity goals; the only effect is that the same continuity goals will be proved more quickly when doing beta reduction.

2.6 EVALUATION

This chapter has described the core of the HOLCF '11 libraries, consisting of the type class hierarchy, various notions of domain theory such as continuity and admissibility, and the definition of all the basic HOLCF types.

The material presented in this chapter was mostly present in some form or other in Regensburger's original HOLCF '95 [Reg94, Reg95]. However, the new HOLCF '11 offers various improvements over the original. The primary novel contributions of this work are threefold: the CPODEF package, the improved automation for continuity proofs, and the use of compactness for proving admissibility.

Cpodef. The Cpodef package provides a very convenient, streamlined way to define new types in HOLCF '11. Using the Cpodef package can result it a big reduction in the size and complexity of proof scripts. For example, in HOLCF '99, the theory of strict products comprised over 900 lines of definitions and proof scripts; after converting the theory to use the Cpodef package, this was reduced to less than 200 lines of code. Similarly, defining the strict sum type with Cpodef reduced that theory file from over 1300 lines to less than 300.

Of all the HOLCF '11 types *not* defined with Cpodef, the lifted cpo type 'a u is the one with the most difficult cpo instance proofs. Unfortunately it seems that no suitable cpo exists from which 'a u could be defined as a subtype, although there is one that comes close:

```
pcpodef 'a u2 = "{p::one × 'a. p = ⊥ ∨ fst p = ONE}"
```

The above definition only works if type 'a is already pointed, however; in contrast, the currently implemented definition works for any 'a in class cpo.

Continuity proofs. The original Edinburgh LCF and Cambridge LCF systems [GMW79, Pau87] did not require proofs of continuity, because reasoning was done *in* the LCF logic, which does not have an explicit concept of continuous functions. In contrast, using HOLCF means reasoning *about* LCF terms and formulas, *in* higher-order logic. Because higher-order logic can express non-continuous functions that are not expressible in LCF, systems like HOLCF that model LCF in higher-order logic must reason explicitly about continuity.

The HOLCF '95 continuity tactic could prove the continuity of any function corresponding to an LCF term. However, for some terms the tactic was prohibitively slow: On nested lambda-abstractions like $(\lambda x.\ \Lambda\ y.\ e\ x\ y)$, the tactic would prove continuity of the body twice, once for each variable. Each additional lambda-abstraction would double the amount of work required, leading to an exponential running time.

HOLCF '95 was not the only formalization of LCF to suffer from this problem. The HOL-CPO system by Sten Agerholm [Age94] is another formalization of LCF in higher-order logic (though implemented in a different theorem prover), released contemporaneously with the original HOLCF '95. In HOL-CPO, continuous function types are represented as subsets of a full function type. Continuity proofs are performed by a "type-checker", which proves that a given function is an element of a particular continuous function type. For abstractions of the form (λx. λy. e x y), the type-checker traverses the body twice, proving continuity separately in x and y [Age94, §5.2.2]. In other words, it works just like the HOLCF '95 continuity tactic: Deeply nested abstractions would cause the same exponential run-time behavior.

The HOL-CPO system does provide a potential workaround: It defines combinators like curry f = (λx y. f (x, y)), with which users can write multi-argument functions without using nested lambdas. For example, the three-argument function (λx y z. e[x, y, z]) could be written instead as curry (curry (λp. e[fst (fst p), snd (fst p), snd p])). Using curry permits continuity proofs with the efficiency of the HOLCF '11 cont2cont rules, at the expense of being more cumbersome to use.

The improved automation for continuity makes HOLCF '11 useful for reasoning about a larger set of programs. In particular, it is now practical to reason about functions with a large number of arguments. For example, beta-reducing a function of a dozen arguments with the old continuity tactic could easily take a minute or more; the new continuity rules can reduce the same function in a fraction of a second. A faster continuity checker also makes larger datatype definitions more practical, because the Domain package performs many beta-reductions in its internal proofs.

Admissibility proofs. In Cambridge LCF, admissibility was not actually formalized in the logic. Rather, fixed point induction was accompanied by a hard-coded syntactic admissibility test [Pau87, page 200]. The test examined both the structure of the formula, and also the chain-finiteness of the types involved. For example, in LCF the predicate $(\lambda \mathsf{x}.\ \mathsf{f\ x} \sqsubseteq \mathsf{y} \longrightarrow \mathsf{g\ x} = \mathsf{h\ x})$ would always pass the admissibility test, while $(\lambda \mathsf{x}.\ \mathsf{f\ x} = \mathsf{y} \longrightarrow \mathsf{g\ x} = \mathsf{h\ x})$ would pass only if either x or y had a chain-finite type. The earlier Edinburgh LCF used a similar test [GMW79, page 77].

HOLCF '99 included some proof automation for admissibility, with roughly the same capabilities as the hard-coded check in Cambridge LCF. The automation comprised a set of structural rules for various connectives, plus a special admissibility tactic that considered chain-finiteness [MNOS99, Mül98]. The structural rules included all those in Fig. 2.3, plus the rule "cont t $\Longrightarrow$ adm $(\lambda \mathsf{x}.\ \mathsf{t\ x} \neq \bot)$". (No rules involving compactness were present, as HOLCF '99 did not have a notion of compactness.) After applying the structural rules, the admissibility tactic would try to solve any remaining subgoals using adm_subst and adm_chfin.

lemma adm_subst: "$[\![$cont f; adm P$]\!] \Longrightarrow$ adm $(\lambda \mathsf{x}.\ \mathsf{P\ (f\ x)})$"
lemma adm_chfin [simp]: "adm $(\lambda(\mathsf{x{::}'a{::}chfin}).\ \mathsf{P\ x})$"

For example, on the goal adm $(\lambda \mathsf{x}.\ \mathsf{f{\cdot}x} = \mathsf{TT} \longrightarrow \mathsf{g{\cdot}x} = \mathsf{h{\cdot}x})$, applying the structural rules would leave the subgoal adm $(\lambda \mathsf{x}.\ \mathsf{f{\cdot}x} \neq \mathsf{TT})$. Then the tactic would apply adm_subst, leaving the goals adm $(\lambda \mathsf{y}.\ \mathsf{y} \neq \mathsf{TT})$ (an instance of adm_chfin, because tr is chain-finite) and cont $(\lambda \mathsf{x}.\ \mathsf{f{\cdot}x})$ (solvable by the continuity tactic).

Agerholm's HOL-CPO also includes a prover for admissibility (he calls it "inclusiveness") with capabilities similar to the HOLCF '99 tactic [Age94, §5.3].

The limitations of the HOLCF '99 admissibillity tactic were noted by Müller with regards to his formalization of I/O automata [Mül98, §10.3.2]. One of the proof scripts in that formalization[9] requires admissibility of the predicate $(\lambda \mathsf{x}.\ \mathsf{f{\cdot}x} \neq \mathsf{nil})$, where nil is a constructor for a recursive list type that is not chain-finite. When the proof script was written, no applicable lemmas were available to assist with

[9] Müller's formalization is included with the Isabelle distribution, in the `HOLCF/IOA` directory.

a proof of admissibility; Müller resorted to declaring an axiom (no doubt with the intention that it be temporary!) rather than unfolding the definition and attempting a manual proof. In HOLCF '11, with the new compactness rules for admissibility in place, the same admissibility goal is solved automatically by the simplifier.

In HOLCF '11, the sophisticated HOLCF '99 tactic for proving admissibility with adm_subst and adm_chfin has been discontinued, because it is no longer necessary. In practice, it was nearly always used in situations where one of the lemmas adm_compact_not_below, adm_neq_compact, or adm_compact_neq from Fig. 2.4 would now apply. These rules are strictly more powerful than the HOLCF '99 admissibility tactic on such subgoals, because they are not limited to chain-finite types.

Chapter 3

RECURSIVE VALUE DEFINITIONS: THE FIXREC PACKAGE

3.1 INTRODUCTION

In Haskell and most other functional programming languages, recursive definitions are supported directly: In the definition of a function, the constant being defined may occur freely on the right-hand side. This is in contrast to a theorem prover like Isabelle, whose primitive definitions are required to be non-recursive in order to guarantee logical soundness.

In Isabelle, a recursive specification of a new constant cannot be used directly as a definition. The specification must be used indirectly: First, the recursive specification must be somehow transformed into a *non*-recursive definition of the constant. Second, this definition can then be used to prove the original specification as a theorem.

Isabelle/HOL already includes a number of packages that mechanize this kind of process for certain kinds of recursive definitions. The Datatype package provides the **primrec** command for defining primitive-recursive functions over datatypes [NPW02]; the Recdef and Function packages define functions that use well-founded recursion [Sli96, Kra10a].

In HOLCF '99, no definition package for general recursive functions existed; such functions could only be defined by explicitly using the domain-theoretic fixed point combinator fix. Users had to derive recursive equations manually, using the theory of least fixed points. This chapter introduces the Fixrec package for Isabelle/HOLCF, which automates this process: Users of HOLCF '11 can now use

Fixrec to formalize Haskell-style recursive function definitions directly.

Contributions. The current implementation of Fixrec derives from the original version by Amber Telfer in 2004 [Tel04]. Since then, many new features have been added by the present author: The current version supports mutual recursion, and curried functions with any number of arguments. It also supports a wider class of patterns: Function definitions can use lazy or strict constructors, including constructors from Isabelle/HOL datatypes; definitions with overlapping patterns are also supported. The package is now suitable for verifying real Haskell programs: Some example case studies using Fixrec, including proofs by least fixed point induction, can be found in Chapter 7.

Overview. The remainder of this chapter starts by describing the features of the Fixrec package, from a user's point of view (§3.2). Then we establish an implementation strategy, showing how to translate lists of function equations into non-recursive definitions by expressing recursion with a fixed point combinator (§3.3) and performing pattern match compilation (§3.4). Next are details of the actual implementation of the Fixrec package (§3.5). Finally, we conclude with a comparison to related work, and directions for future work (§3.6).

3.2 FIXREC PACKAGE FEATURES

The Fixrec package lets users define recursive functions and values in HOLCF much like they can in Haskell or ML. Users can write function specifications with patterns, involving datatypes that may have been defined by the user. Users can use recursion freely within groups of simultaneously-defined values, with no need for termination proofs. The Fixrec package generates definitions for constants, proves each defining equation as a theorem, and also generates induction rules for reasoning about the new constants.

Patterns in Fixrec definitions can mention any of the constructors for basic HOLCF types (§2.4): spair, sinl, sinr, up, ONE, TT, FF, and also the Pair constructor (_, _) from Isabelle/HOL. Constructors for types defined by the Domain package are also supported, for instance LNil and LCons from the lazy list type below.

```
domain 'a llist = LNil | LCons (lazy "'a") (lazy "'a llist")
```

The syntax for the **fixrec** command is similar to other definition commands like **primrec** or the Function package's **fun** command. (A full syntax diagram is shown in Fig. 3.1.) The equations must be separated by vertical bars; theorem names for each equation are optional.

```
fixrec firsts :: "('a × 'b) llist → 'a llist"
  where firsts_LNil: "firsts·LNil = LNil"
  | firsts_LCons: "firsts·(LCons·(x, y)·xs) = LCons·x·(firsts·xs)"
```

The definition above generates the theorems firsts_LNil and firsts_LCons. It also declares a theorem list firsts.simps consisting of all equations in the definition; these are added to the simplifier by default.

Lazy and strict constructors. The Fixrec package works best with lazy constructor functions like LCons, Pair and up. It also works with strict constructor functions, but definedness side conditions like $x \neq \bot$ may be required.

```
fixrec from_sinl :: "'a ⊕ 'b → 'a"
  where "x ≠ ⊥ ⟹ from_sinl·(sinl·x) = x"
```

If the side-condition in the above definition is omitted, the Fixrec package will not be able to prove the equation, and it will fail with an error message. Note that combinations of strict and lazy constructors can avoid the need for such side conditions.

```
fixrec from_sinl_up :: "'a⊥ ⊕ 'b → 'a"
  where "from_sinl_up·(sinl·(up·x)) = x"
```

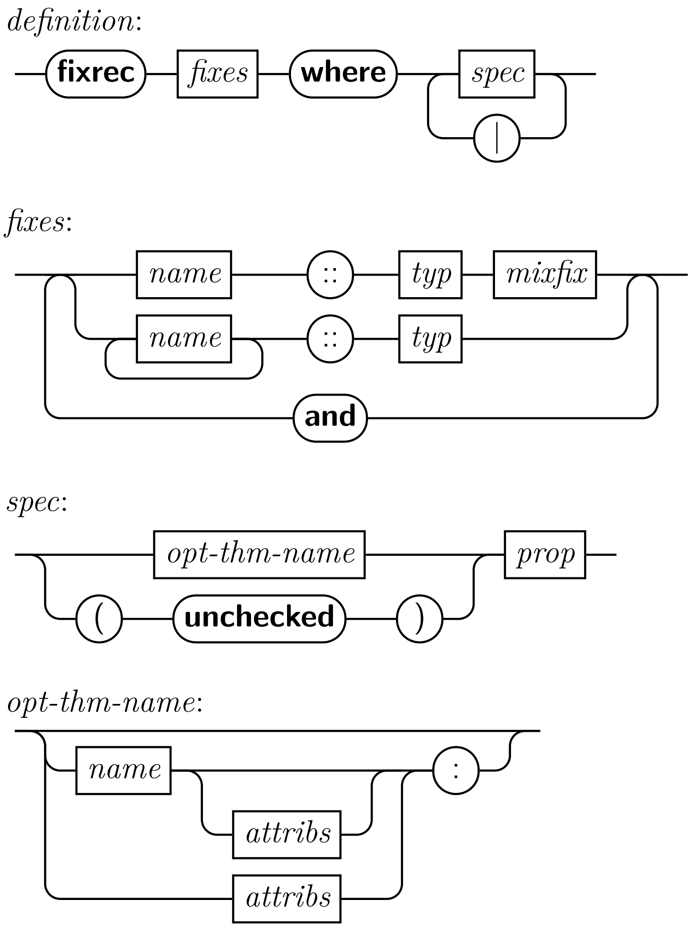


Figure 3.1: Input syntax for FIXREC package

Even though sinl is a strict constructor, the side condition up·x ≠ ⊥ is not needed. This is because up is a lazy constructor and Fixrec can determine up·x ≠ ⊥ automatically.

Overlapping patterns. The Fixrec package supports definitions with overlapping patterns. For example, consider the following function that zips two lists together: The first equation has specific patterns, while the second equation is a catch-all default case. The second equation cannot be proved as a theorem because it only applies when the first pattern fails.

```
fixrec lzip :: "'a llist → 'b llist → ('a × 'b) llist"
  where "lzip·(LCons·x·xs)·(LCons·y·ys) = LCons·(x, y)·(lzip·xs·ys)"
  | (unchecked) "lzip·xs·ys = LNil"
```

Usually Fixrec tries to prove all equations as theorems. The **unchecked** option overrides this behavior, so Fixrec does not attempt to prove that particular equation. With the lzip example, both equations influence the definition of lzip, but only the first is proved as a theorem. The generated theorem list lzip.simps then consists of only the first equation.

Generating extra equations. The Fixrec package provides automation for proving extra equations beyond those included in the function definition. This takes the form of a proof method called fixrec_simp. One use for fixrec_simp is to prove specific rules for functions defined with overlapping patterns, like our earlier example lzip.

```
lemma lzip_extra_simps [simp]:
  shows "lzip·(LCons·x·xs)·LNil = LNil" and "lzip·LNil·ys = LNil"
  by fixrec_simp+
```

Another common use of the fixrec_simp method is to prove strictness rules for functions.

```
lemma lzip_strict [simp]:
  shows "lzip·⊥·ys = ⊥" and "lzip·(LCons·x·xs)·⊥ = ⊥"
  by fixrec_simp+
```

Functions defined by Fixrec satisfy the same strictness properties that we would find in Haskell, according to a left-to-right pattern matching strategy. For example, lzip·⊥·LNil evaluates to ⊥, but the opposite argument order yields a different result: lzip·LNil·⊥ = LNil.

Non-exhaustive patterns. Unlike other definition packages in Isabelle, the Fixrec package does not require patterns to be exhaustive.

```
fixrec lzip2 :: "'a llist → 'b llist → ('a × 'b) llist"
  where "lzip2·(LCons·x·xs)·(LCons·y·ys) = LCons·(x, y)·(lzip·xs·ys)"
  | "lzip2·LNil·LNil = LNil"
```

If none of the equations match, the result for those arguments defaults to ⊥. Again, fixrec_simp is useful for generating the additional theorems.

```
lemma lzip2_LCons_LNil: "lzip2·(LCons·x·xs)·LNil = ⊥"
  by fixrec_simp
```

Induction rules. The Fixrec package generates a specialized fixed point induction rule for each recursive definition. For example, consider this recursively-defined while combinator:[1]

```
fixrec while :: "('a → tr) → ('a → 'a) → 'a → 'a"
  where [simp del]: "while·p·f·x = (If p·x then while·p·f·(f·x) else x)"
```

To prove properties about while, we can use the induction rule while.induct provided by Fixrec. First, the predicate P must be admissible. For the base case, P must hold for ⊥. For the inductive step, we assume that P holds for an arbitrary function

[1] The attribute [simp del] prevents the equation being added to the simplifier; this is desirable because otherwise it would cause the simplifier to loop.

w, and then show that it must also hold for an unfolded version of the while function where recursive calls are replaced by calls to w.

```
theorem while.induct:
  "⟦adm P; P ⊥; ⋀w. P w ⟹ P (Λ p f x. If p·x then w·p·f·(f·x) else x)⟧
    ⟹ P while"
```

Note that the induction rules generated by Fixrec are somewhat unusual compared to those generated by Recdef or the Function package. In those packages, the induction rule for a function uses a predicate on the *function's arguments*—in Fixrec, the predicate P in the induction rule is a predicate on the *function itself*. It is essentially like doing induction over the number of times the function's definition is unfolded.

As an example, we can use fixed point induction to prove the following property of while: Either the predicate p evaluates to false on the result, or else the computation diverges.

```
lemma while_post_condition: "∀x. p·(while·p·f·x) = FF ∨ while·p·f·x = ⊥"
  apply (rule while.induct) ...
```

In this example, we use the fixed point induction rule while.induct with the predicate P instantiated to $(\lambda w.\ \forall x.\ p\cdot(w\cdot p\cdot f\cdot x) = FF \lor w\cdot p\cdot f\cdot x = \bot)$. Admissibility of P and the base case $P\ \bot$ can be solved automatically by the Isabelle simplifier, while the inductive step requires a case analysis on $p\cdot x$.

The definition of while uses only variable patterns, so its fixed point induction rule looks relatively simple. Fixrec generates a similar (but more complicated-looking) rule when a function is defined with pattern matching. In this case, a compiled version of the patterns will appear explicitly in the inductive step. (More details will be given in Sec. 3.5.4.)

Mutual recursion. The Fixrec package can define multiple functions simultaneously, where each one can call any of the others. To define mutually recursive functions, give multiple type signatures separated by the keyword **and**.

```
fixrec evenlen :: "'a llist → tr" and oddlen :: "'a llist → tr"
  where "evenlen·LNil = TT"
  | "evenlen·(LCons·x·xs) = oddlen·xs"
  | "oddlen·LNil = FF"
  | "oddlen·(LCons·x·xs) = evenlen·xs"
```

For mutually recursive definitions, Fixrec produces a single fixed point induction rule that can be used to prove properties of all the recursive constants simultaneously. For example, the definition of evenlen and oddlen above yields the rule evenlen_oddlen.induct, where P is a binary predicate of both constants. (The inductive step contains a rather large pattern-match-compiled version of both function bodies, which we elide here; see Sec. 3.5.6 for the full details.)

```
theorem evenlen_oddlen.induct:
  "⟦adm (λ(a, b). P a b); P ⊥ ⊥; ⋀a b. P a b ⟹ P ... ...⟧
    ⟹ P evenlen oddlen"
```

We can use this rule to prove, for example, that evenlen diverges if and only if oddlen also diverges on the same argument.

```
lemma evenlen_bottom_iff: "∀xs. evenlen·xs = ⊥ ⟷ oddlen·xs = ⊥"
  apply (rule evenlen_oddlen.induct) ...
```

In this proof we use the induction rule evenlen_oddlen.induct with the predicate P instantated to (λa b. ∀xs. a·xs = ⊥ ⟷ b·xs = ⊥). The admissibility condition and the base case are both proved automatically, and the inductive step proceeds by case analysis on xs.

3.3 EXPRESSING RECURSION WITH FIX

In the context of domain theory, we can use the least fixed point combinator to transform a recursive specification into a non-recursive one. The details of this transformation will be demonstrated using examples in Haskell. The first example is a simple recursive value that does not take any function arguments; it defines an infinite list.

```
trues :: [Bool]
trues = True : trues
```

To translate this equation to a non-recursive definition, we abstract the right-hand side over all occurrences of the constant being defined, and apply `fix` to the resulting function:

```
fix :: (a -> a) -> a
fix f = f (fix f)

trues = fix (\r -> True : r)
```

The next example is similar, but adds a function argument:

```
repeat :: a -> [a]
repeat x = x : repeat x
```

To translate the equation for `repeat`, we first convert the function argument pattern into a lambda abstraction. Then we proceed as before, abstracting over recursive occurrences of `repeat`, and applying `fix`.

```
repeat = \x -> x : repeat x
repeat = fix (\r x -> x : r x)
```

The third example adds another new feature: multiple equations with patterns. This function calculates the sum of a list of integers.

```
sum :: [Int] -> Int
sum [] = 0
sum (x : xs) = x + sum xs
```

The patterns of the `sum` function require yet another translation step. We start by converting the set of pattern-match equations to a single equation with a case expression. The rest of the translation proceeds as before.

```
sum ys = case ys of [] -> 0; x : xs -> x + sum xs
sum = \ys -> case ys of [] -> 0; x : xs -> x + sum xs
sum = fix (\r ys -> case ys of [] -> 0; x : xs -> x + r xs)
```

The final example shows how mutually recursive definitions can be translated using `fix`. Below we have a list of equations, one for each mutually defined constant. (In general, we expect that any function arguments or patterns should have been translated away by this point.)

```
list1, list2 :: [Bool]
list1 = True : list2
list2 = False : list1
```

The next step is to use tuples to combine all the equations into one. Now we can define the tuple of all the new constants using `fix`, where we use the projections `fst` and `snd` to refer to occurrences of either constant on the right-hand side.

```
(list1, list2) = (True : list2, False : list1)
(list1, list2) = fix (\r -> (True : snd r, False : fst r))
```

Finally we define each individual constant by projecting out the components of the fixed point.

```
list1 = fst (fix (\r -> (True : snd r, False : fst r)))
list2 = snd (fix (\r -> (True : snd r, False : fst r)))
```

In summary, the translation from function equations to a fixed point definition consists of the following four steps:

1. Compile patterns to case-expressions
2. Convert function arguments to lambdas
3. Abstract over recursive calls, and apply `fix`
4. Project components of fixed point tuple (if necessary)

The later sections of this chapter will describe how the Fixrec package automates all of these steps. Before getting into the implementation details of the package, we will first describe the approach to pattern-match compilation that Fixrec uses.

```
oddfsts :: [(a, b)] -> [a]
oddfsts ((a, b) : y : zs) = a : oddfsts zs
oddfsts [(a, b)] = [a]
oddfsts [] = []
```

Figure 3.2: A Haskell function definition with nested patterns

3.4 PATTERN MATCH COMPILATION

The pattern-matching example in the previous section used only simple patterns like `[]` and `x : xs`. The meaning of functions with such patterns is relatively straightforward. But things get more complicated when functions use *nested* patterns, where constructors are applied to one or more sub-patterns instead of just variables. Figure 3.2 shows an example of a Haskell function using nested patterns, which returns the first component of every other element from a list of pairs.

Pattern match equations with nested patterns can get rather complex. We will specify the meaning of a list of pattern match equations by "compiling" them down to a combination of simpler building blocks. This pattern match compilation can be implemented in various ways. The remainder of this section will describe a couple of alternative approaches, finishing with the specific design currently used by the FIXREC package.

3.4.1 Compiling to simple case expressions

One possible approach to pattern match compilation is to convert patterns into combinations of *simple* case expressions (i.e. those without nested patterns). A simple case expression for any given datatype can be written using a case combinator function; combinators for lists and pairs are defined as Haskell functions in Fig. 3.3.

Figure 3.4 shows the function `oddfsts` after the patterns have been compiled

```
listcase :: b -> (a -> [a] -> b) -> [a] -> b
listcase z f [] = z
listcase z f (x : xs) = f x xs

paircase :: (a -> b -> c) -> (a, b) -> c
paircase f (x, y) = f x y
```

Figure 3.3: Combinators for simple case expressions

```
oddfsts xs = case xs of
               []     -> []
               x : ys -> case x of
                           (a, b) -> case ys of
                                       []     -> [a]
                                       y : zs -> a : oddfsts zs

oddfsts xs =
  listcase [] (\x ys ->
    paircase (\a b ->
      listcase [a] (\y zs -> a : oddfsts zs) ys) x) xs
```

Figure 3.4: Function compiled to simple case expressions, with equivalent case combinators

down to simple case expressions. It also shows an equivalent definition expressed using the `listcase` and `paircase` combinators.

Algorithms for doing this style of pattern-match compilation are described in the literature [Wad87]. Such algorithms are implemented as part of various compilers for functional programming languages, and also in existing Isabelle packages like Recdef [Sli96].

This style of pattern-match compilation has some desirable properties. One benefit is the efficiency of the compiled code: The case expressions never analyze the same subterm more than once. Also, simple case expressions are convenient

```
data Maybe a = Nothing | Just a

(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
Just x >>= k = k x
Nothing >>= k = Nothing

(+++) :: Maybe a -> Maybe a -> Maybe a
Just x +++ y = Just x
Nothing +++ y = y

run :: Maybe a -> a
run (Just x) = x
run Nothing = undefined
```

Figure 3.5: Maybe monad with fatbar and run operators

to use in Isabelle, because appropriate case combinators are already provided for each datatype. There are also some drawbacks, however. In definitions with one or more specific pattern equations followed by a catch-all default case, the compiled terms can get big, and cleverness is required to avoid duplicating case branches [Wad87, §5.4.1]. Recdef sidesteps this issue by disallowing overlapping patterns; however, overlapping patterns are a supported feature of Fixrec (see Sec. 3.2). To avoid bugs, it is preferable to have an implementation of pattern matching that is as simple as possible.

3.4.2 Original Fixrec: Monadic pattern matching

This section describes the system used by Telfer's original implementation of Fixrec [Tel04]. Inspired by a monadic-style semantics of pattern matching in Haskell [HSH02], it uses the `Maybe` type (see Fig. 3.5) to model the possibility of pattern-match failure.

Each equation in the function definition is treated independently. When the patterns of an equation are matched against a list of arguments, there are three possible outcomes, each of which can be represented in the `Maybe` type: A match can either succeed (`Just x`), fail (`Nothing`), or diverge (⊥).

The results of multiple pattern-match equations are combined using the `(+++)` operator (written as ▯ and called "fatbar" by by Peyton Jones and Wadler [PJ87], and also implemented as `mplus` in the Haskell standard libraries). The result of `m1 +++ m2 +++ ... +++ mn` equals the first value in the list different from `Nothing`. It is straightforward to verify that `(+++)` is associative.

After the results of all the pattern match equations are combined with the fatbar operator, we can extract the value of the first successful pattern match using the `run` function. If none of the pattern matches are successful (i.e., all equations evaluate to `Nothing`), then the result is `undefined` or ⊥.

In addition to the operators shown in Fig. 3.5, the original Fixrec implementation also required one *match combinator* for each constructor that might be used in patterns. The match combinator for a constructor examines its input value, and if the constructor matches, returns `Just` applied to a tuple of the constructor's arguments; for any other constructor it returns `Nothing`. Haskell definitions of some match combinators are given in Fig. 3.6.

To compile a pattern match equation, we start by inventing a fresh variable name for each sub-pattern. Then we traverse each pattern in a top-down, left-to-right manner. We produce the corresponding match combinator for each constructor in the pattern, sequencing the combinators using the bind operator `(>>=)`.

Figure 3.7 shows the result of compiling the function `oddfsts` using the monadic match combinators. For readability, the monadic terms are shown using Haskell's *do* syntax: `do {x <- m; k}` stands for `m >>= (\x -> k)`, and `do {a; b; c}` means `do {a; do {b; c}}`.

For implementing a pattern match compiler, the monadic match combinators offer some benefits compared to simple case combinators. The primary advantage is

```
mNil :: [a] -> Maybe ()
mNil [] = Just ()
mNil (x : xs) = Nothing

mCons :: [a] -> Maybe (a, [a])
mCons [] = Nothing
mCons (x : xs) = Just (x, xs)

mPair :: (a, b) -> Maybe (a, b)
mPair (x, y) = Just (x, y)
```

Figure 3.6: Monadic match combinators like those used by original Fixrec

```
oddfsts xs = run (
  do { (x, ys) <- mCons xs;
       (a, b) <- mPair x;
       (y, zs) <- mCons ys;
       Just (a : oddfsts zs) }
  +++
  do { (x, ys) <- mCons xs;
       (a, b) <- mPair x;
       () <- mNil ys;
       Just [a] }
  +++
  do { () <- mNil xs;
       Just [] } )
```

Figure 3.7: A function compiled using monadic match combinators

ease of implementation: The monadic pattern match compiler is a straightforward algorithm that can be implemented with a small amount of ML code, making a single traversal of the syntax tree of the pattern. Another advantage is that the compiled output has a predictable size: The number of match combinators in the compiled function is always equal to the number of constructors in the input patterns; in general, the size of the output is always in linear proportion to the size of the input. In comparison, compilation to simple case combinators may sometimes yield smaller terms—such as with the example function `oddfsts`—but some patterns can yield large output with repeated sub-terms, unless optimizations are added to handle such cases [Wad87, §5.4.1].

One potential drawback of monadic pattern match compilation is that the `run` function requires the final result type to have an `undefined` or $\perp$ value in case none of the patterns match—even if the patterns are in fact complete. Unlike with compilation to case combinators, monadic pattern compilation does not offer an easy way to verify the completeness of patterns. This may be an important concern for packages that define total functions, such as RECDEF or the Isabelle FUNCTION package. However it is not a issue for FIXREC, because the least fixed point combinator `fix` already requires the return type to have a bottom element.

3.4.3 New Fixrec: Continuation-based matching combinators

The monadic pattern matching system described above is workable—indeed, it was sufficient for implementing the original FIXREC package—but it leaves room for improvement. Specifically, the compiled terms are bigger than they need to be. Making some simple changes to the definitions of the match combinators will allow the pattern match compiler to produce equivalent, but smaller output.

Note that in the compiled monadic term in Fig. 3.7, match combinators like `mCons` always occur in combination with the monadic bind operator `(>>=)` and a tuple binding. Accordingly, we can define new match combinators that have this

```
matNil xs k = do { () <- mCons xs; k }
matCons xs k = do { (a, b) <- mCons xs; k a b }
matPair xs k = do { (a, b) <- mPair xs; k a b }
```

Figure 3.8: Specification of continuation-based match combinators

```
matNil :: [a] -> Maybe b -> Maybe b
matNil [] k = k
matNil (x : xs) k = Nothing

matCons :: [a] -> (a -> [a] -> Maybe b) -> Maybe b
matCons [] k = Nothing
matCons (x : xs) k = k x xs

matPair :: (a, b) -> (a -> b -> Maybe c) -> Maybe c
matPair (x, y) k = k x y
```

Figure 3.9: Definition of continuation-based combinators used by new FIXREC package

additional functionality built in. Figure 3.8 specifies these new combinators in terms of the old monadic ones. Figure 3.9 gives the direct definitions—note that they are just as simple as the old monadic combinators (Fig. 3.6).

The new combinators each take an extra *continuation* argument, representing the remainder of the compiled pattern-matching expression. With continuation-based combinators, bind operators and tuples are no longer needed to compile patterns. Figure 3.10 shows the example function `oddfsts` expressed in these new combinators. In comparison, the monadic compiled term in Fig. 3.7 may appear to have a similar size, but unfolding the syntactic sugar (*do*-notation and tuple bindings) reveals that in terms of the actual number of constants, the new term is significantly smaller.

```
oddfsts xs = run (
  matCons xs (\x ys -> matPair x (\a b ->
    matCons ys (\y zs -> Just (a : oddfsts zs))))
  +++
  matCons xs (\x ys -> matPair x (\a b -> matNil ys (Just [a])))
  +++
  matNil xs (Just []) )
```

Figure 3.10: A function compiled using the continuation-based match combinators

3.5 IMPLEMENTATION

The implementation of the FIXREC package consists of two main parts. First, an Isabelle theory file contains various supporting definitions and lemmas, including the pattern match type with its associated operations (§3.5.1) and match combinators for the basic HOLCF types (§3.5.2). The remainder of FIXREC is implemented as ML code, which performs pattern match compilation (§3.5.3), generates fixed point definitions (§3.5.4), and proves equations (§3.5.5). Some parts of the implementation have specific features for handling mutual recursion (§3.5.6).

3.5.1 Pattern match type

The FIXREC theory defines a type 'a match to model the Haskell `Maybe` monad. Normally, the DOMAIN package would be the natural way to define such a datatype in HOLCF:

domain 'a match = fail | succeed (**lazy** 'a)

However, due to the bootstrapping order of HOLCF, the DOMAIN package is not available at this point. Instead, we use the CPODEF package (§2.3) to define type 'a match as an isomorphic copy of $\mathsf{one} \oplus \mathsf{'a}_{\perp}$. Next, the constructors for type 'a match are defined manually, in the same manner as they would have been defined by the DOMAIN package (see Fig. 3.11).

```
pcpodef 'a match = "UNIV :: (one ⊕ 'a⊥) set"

definition fail :: "'a match"
  where "fail = Abs_match (sinl·ONE)"

definition succeed :: "'a → 'a match"
  where "succeed = (Λ x. Abs_match (sinr·(up·x)))"

definition run :: "'a match → 'a::pcpo"
  where "run = (Λ m. sscase·⊥·(fup·ID)·(Rep_match m))"

definition mplus :: "'a match → 'a match → 'a match"
  where "mplus = (Λ m1 m2. sscase·(Λ x. m2)·(Λ x. m1)·(Rep_match m1))"
```

Figure 3.11: Definitions of pattern match type and associated functions

The run and mplus functions are defined using sscase and fup, the case combinators for the strict sum and lifted cpo types (§2.4). The essential properties of these functions are that they satisfy the rewrite rules below.[2]

```
lemma run_simps [simp]:
  shows "run·⊥ = ⊥" and "run·fail = ⊥" and "run·(succeed·x) = x"

lemma mplus_simps [simp]:
  shows "mplus·⊥·m = ⊥" and "mplus·fail·m = m"
    and "mplus·(succeed·x)·m = succeed·x"
```

3.5.2 Table of pattern match combinators

Using Isabelle's theory data mechanism [WW07], FIXREC maintains a table that maps each constructor function to the name of its corresponding pattern match combinator. Each pattern match combinator must have a specific two-argument function type, based on the type of its constructor. The first argument has the

[2] A possible alternative design would be to define 'a match = "'a → 'a", fail = ID, succeed·x = (Λ a. x), run·m = m·⊥, and mplus·m1·m2 = (Λ a. m1·(m2·a)), because these operations satisfy the same rules.

result type of the constructor function. The second argument of the combinator is a continuation function, taking a list of the constructor's arguments and returning a match type. For example, a constructor LCons :: 'a → 'a llist → 'a llist might have a combinator match_LCons :: 'a llist → ('a → 'a llist → 'b match) → 'b match.

Constructors with HOL function types are also allowed. The Isabelle/HOL constructor Pair, which has type 'a ⇒ 'b ⇒ 'a × 'b, has a corresponding match combinator match_Pair :: 'a × 'b → ('a → 'b → 'c match) → 'c match. Fixrec expects match combinators to use the continuous function space (→) throughout, regardless of which function type (⇒ vs. →) the constructor uses. It might be beneficial to lift this restriction, because the current system does not support constructors like Def :: 'a ⇒ 'a lift, which have argument types that are not cpos. However, doing so would require modifications to both the combinator table and the pattern match compiler, and has not been implemented.

In its initial configuration, the combinator table includes entries for all the constructors of the basic types used in HOLCF: pairs (Pair and spair), strict sum (sinl and sinr), lifting (up), and flat unit and boolean types (ONE, TT and FF). The match combinators for these are defined manually, as shown in Fig. 3.12. The rules shown in Fig. 3.13 are declared as default simplification rules. The Domain package also generates match combinators for each new datatype, and adds them to the table (see Chapter 4).

3.5.3 Pattern match compilation

This next few subsections will use an example function definition to demonstrate the inner workings of the Fixrec package.

```
domain 'a llist = LNil | LCons (lazy 'a) (lazy "'a llist")

fixrec firsts :: "('a × 'b) llist → 'a llist"
  where "firsts·LNil = LNil" | "firsts·(LCons·(x, y)·xs) = LCons·x·(firsts·xs)"
```

```
definition match_Pair :: "'a × 'b → ('a → 'b → 'c match) → 'c match"
  where "match_Pair = (Λ x k. csplit·k·x)"

definition match_spair :: "'a ⊗ 'b → ('a → 'b → 'c match) → 'c match"
  where "match_spair = (Λ x k. ssplit·k·x)"

definition match_sinl :: "'a ⊕ 'b → ('a → 'c match) → 'c match"
  where "match_sinl = (Λ x k. sscase·k·(Λ b. fail)·x)"

definition match_sinr :: "'a ⊕ 'b → ('b → 'c match) → 'c match"
  where "match_sinr = (Λ x k. sscase·(Λ a. fail)·k·x)"

definition match_up :: "'a⊥ → ('a → 'c match) → 'c match"
  where "match_up = (Λ x k. fup·k·x)"

definition match_ONE :: "one → 'c match → 'c match"
  where "match_ONE = (Λ ONE k. k)"

definition match_TT :: "tr → 'c match → 'c match"
  where "match_TT = (Λ x k. If x then k else fail)"

definition match_FF :: "tr → 'c match → 'c match"
  where "match_FF = (Λ x k. If x then fail else k)"
```

Figure 3.12: Definitions of pattern match combinators for basic HOLCF types

```
lemma match_Pair_simps [simp]:
  "match_Pair·(x, y)·k = k·x·y"

lemma match_spair_simps [simp]:
  "match_spair·⊥·k = ⊥"
  "⟦x ≠ ⊥; y ≠ ⊥⟧
    ⟹ match_spair·(:x, y:)·k = k·x·y"

lemma match_sinl_simps [simp]:
  "match_sinl·⊥·k = ⊥"
  "x ≠ ⊥ ⟹ match_sinl·(sinl·x)·k = k·x"
  "y ≠ ⊥ ⟹ match_sinl·(sinr·y)·k = fail"

lemma match_sinr_simps [simp]:
  "match_sinr·⊥·k = ⊥"
  "x ≠ ⊥ ⟹ match_sinr·(sinl·x)·k = fail"
  "y ≠ ⊥ ⟹ match_sinr·(sinr·y)·k = k·y"
```

```
lemma match_up_simps [simp]:
  "match_up·⊥·k = ⊥"
  "match_up·(up·x)·k = k·x"

lemma match_ONE_simps [simp]:
  "match_ONE·⊥·k = ⊥"
  "match_ONE·ONE·k = k"

lemma match_TT_simps [simp]:
  "match_TT·⊥·k = ⊥"
  "match_TT·TT·k = k"
  "match_TT·FF·k = fail"

lemma match_FF_simps [simp]:
  "match_FF·⊥·k = ⊥"
  "match_FF·FF·k = k"
  "match_FF·TT·k = fail"
```

Figure 3.13: Simplification rules for pattern match combinators

The first step that Fixrec needs to do is pattern match compilation. As input, the pattern match compiler gets the list of equations provided by the user; on the left-hand side of each equation is the function being defined, applied to some number of patterns. The output of the pattern match compiler will be a single equation with no patterns.

Each equation is compiled separately to a function that returns a match result. So for our example function firsts :: ('a × 'b) llist → 'a llist, each equation will be compiled to a function of type ('a × 'b) llist → 'a llist match.

To compile a single equation, the pattern match compiler traverses the patterns in a bottom-up, right-to-left manner, using an accumulating parameter to incrementally build up the result. We will examine the steps taken while compiling the second pattern match equation:

$$\mathsf{firsts}\cdot(\mathsf{LCons}\cdot(\mathsf{x},\ \mathsf{y})\cdot\mathsf{xs}) = \mathsf{LCons}\cdot\mathsf{x}\cdot(\mathsf{firsts}\cdot\mathsf{xs})$$

The accumulating parameter is initially just succeed applied to the right-hand side:

$$\mathsf{succeed}\cdot(\mathsf{LCons}\cdot\mathsf{x}\cdot(\mathsf{firsts}\cdot\mathsf{xs}))$$

Going bottom-up and right-to-left, the first pattern to process is a variable pattern, xs. For variable patterns, the accumulating parameter does not change; the algorithm simply notes the name of the variable and continues to the next sub-pattern. The variable sub-patterns y and x are handled next.

Now the algorithm moves up to the constructor pattern, (x, y). For a constructor pattern, we perform the following steps:

1. Choose a fresh variable name (v) for the pattern
2. Look up the constructor (Pair) in the table to get the combinator (match_Pair)
3. Abstract over the variables from the sub-patterns (x and y) to construct a continuation
4. Apply the combinator to the fresh variable and the continuation

After processing the constructor pattern (x, y), the accumulating parameter now has this value:

$$\mathsf{match\_Pair}\cdot\mathsf{v}\cdot(\Lambda\ \mathsf{x}\ \mathsf{y}.\ \mathsf{succeed}\cdot(\mathsf{LCons}\cdot\mathsf{x}\cdot(\mathsf{firsts}\cdot\mathsf{xs})))$$

The LCons constructor pattern is then processed similarly; it uses the fresh variable name a, resulting in the following term.

$$\mathsf{match\_LCons}\cdot\mathsf{a}\cdot(\Lambda\ \mathsf{v}\ \mathsf{xs}.\ \mathsf{match\_Pair}\cdot\mathsf{v}\cdot(\Lambda\ \mathsf{x}\ \mathsf{y}.\ \mathsf{succeed}\cdot(\mathsf{LCons}\cdot\mathsf{x}\cdot(\mathsf{firsts}\cdot\mathsf{xs}))))$$

At this point, if there were any other function arguments, we would process each of them in sequence from right to left. The final result of compiling a single equation then consists of the compiled pattern, together with the list of variable names from the top-level patterns. In this case, there is only one such variable (a) because the function firsts only takes one argument. The result of compiling the first equation is shown below; it also has just one top-level variable (a).

$$\mathsf{match\_LNil}\cdot\mathsf{a}\cdot(\mathsf{succeed}\cdot\mathsf{LNil})$$

After all the equations have been processed, the final step is to combine all the compiled patterns using the mplus and run operators, and abstract over the list of variables from the top-level patterns. In preparation for this step, it may be necessary to do a variable name substitution, to ensure that each equation uses the same variable names. Below is the final combined result of compiling both equations.

$$\begin{aligned}&\mathsf{firsts} = (\Lambda\ \mathsf{a}.\ \mathsf{run}\cdot(\mathsf{match\_LNil}\cdot\mathsf{a}\cdot(\mathsf{succeed}\cdot\mathsf{LNil}) +\!\!+\!\!+\\ &\quad \mathsf{match\_LCons}\cdot\mathsf{a}\cdot(\Lambda\ \mathsf{v}\ \mathsf{xs}.\ \mathsf{match\_Pair}\cdot\mathsf{v}\cdot(\Lambda\ \mathsf{x}\ \mathsf{y}.\ \mathsf{succeed}\cdot(\mathsf{LCons}\cdot\mathsf{x}\cdot(\mathsf{firsts}\cdot\mathsf{xs}))))))\end{aligned}$$

3.5.4 Fixed point definition and continuity proof

After pattern match compilation, the next step is to create a *functional* by abstracting the right-hand side over all occurrences of the constant being defined. Fixrec defines the constant as the least fixed point of this functional.

```
definition firsts_def:
  "firsts ≡ fix·(Λ f a. run·(match_LNil·a·(succeed·LNil) +++
    match_LCons·a·(Λ v xs. match_Pair·v·(Λ x y. succeed·(LCons·x·(f·xs))))))"
```

To ensure that the least fixed point exists, and that firsts is indeed a least fixed point, it is necessary for FIXREC to prove that the functional is continuous. Internally, FIXREC creates a goal state and attempts to prove the following continuity lemma.

```
have firsts.cont:
  "cont (λ f. Λ a. run·(match_LNil·a·(succeed·LNil) +++
    match_LCons·a·(Λ v xs. match_Pair·v·(Λ x y. succeed·(LCons·x·(f·xs))))))"
```

The proof proceeds by applying the tactic intro cont2cont (see Sec. 2.5). If that tactic fails to completely solve the goal, then we back up and try the Isabelle simplifier instead. If neither tactic can solve the goal, then FIXREC aborts with an error message. In our example, the functional uses only continuous application and abstraction, so the rules in cont2cont are sufficient to complete the proof.

FIXREC uses the theorems firsts_def and firsts.cont to derive a couple of other theorems. The first is the *unfold rule*, which states that firsts is indeed a fixed point of the appropriate functional. The theorem firsts.unfold is produced by resolving firsts_def and firsts.cont with the library lemma def_cont_fix_eq. Recall that (Λ x. e) is syntactic sugar for Abs_cfun (λ x. e), which explains the presence of Abs_cfun in the lemma.

```
lemma def_cont_fix_eq:
  "⟦f ≡ fix·(Abs_cfun F); cont F⟧ ⟹ f = F f"

theorem firsts.unfold:
  "firsts = (Λ a. run·(match_LNil·a·(succeed·LNil) +++
    match_LCons·a·(Λ v xs. match_Pair·v·(Λ x y. succeed·(LCons·x·(firsts·xs))))))"
```

The next derived theorem is the *induction rule*, which basically states that firsts is the *least* fixed point of its functional. Like firsts.unfold, theorem firsts.induct is also derived from firsts_def and firsts.cont, but now using lemma def_cont_fix_ind.

```
lemma def_cont_fix_ind:
  "⟦f ≡ fix·(Abs_cfun F); cont F; adm P; P ⊥; ⋀x. P x ⟹ P (F x)⟧ ⟹ P f"

theorem firsts.induct:
  "⟦adm P; P ⊥;
    (⋀x. P x ⟹ P (Λ a. run·(match_LNil·a·(succeed·LNil) +++
      match_LCons·a·(Λ v xs. match_Pair·v·(Λ x' y. succeed·(LCons·x'·(x·xs)))))))⟧
    ⟹ P firsts"
```

Note that the pattern matching combinators appear explicitly in the induction rule. Perhaps it would be preferable to hide such implementation details from users, but it is unclear how best to accomplish this. A possibility would be to have Fixrec define the functional as a named constant, which would make the induction rules more concise. But in proofs by induction users would have to unfold the functional's definition, exposing them once more to the matching combinators. Further directions for improvement will be discussed in this chapter's conclusion.

3.5.5 Proving pattern match equations

This section will show how Fixrec uses unfold rules and the simplifier to prove the original defining equations as theorems. Fixrec sets up a goal state for each defining equation, and tries to prove each one in turn. We will examine the proof of the second equation, as it is more interesting.

```
theorem "firsts·(LCons·(x, y)·xs) = LCons·x·(firsts·xs)"
```

The first step in the proof is to substitute the unfold lemma firsts.unfold on the left-hand side only. This leaves the following goal state:

```
goal (1 subgoal):
 1. (Λ a. run·(match_LNil·a·(succeed·LNil) +++ match_LCons·a·(Λ v xs.
    match_Pair·v·(Λ x y. succeed·(LCons·x·(firsts·xs))))))·(LCons·(x, y)·xs)
  = LCons·x·(firsts·xs)
```

The next step is to apply the simplifier (**apply** simp). If the simplifier is not able to prove the goal, then Fixrec will abort with an error message. In this example, the simplifier will be able to solve the goal, assuming that the following lemmas about the match combinators have been added to the simplifier:

```
"match_LNil·⊥·k = ⊥"                "match_LCons·⊥·k = ⊥"
"match_LNil·LNil·k = k"             "match_LCons·LNil·k = fail"
"match_LNil·(LCons·x·xs)·k = fail"  "match_LCons·(LCons·x·xs)·k = k·x·xs"
```

We will examine the actions of the simplifier to illustrate how these simp rules are used. The first thing the simplifier does is beta-reduction, leaving the subgoal below.

```
goal (1 subgoal):
 1. run·(match_LNil·(LCons·(x, y)·xs)·(succeed·LNil) +++ match_LCons·
      (LCons·(x, y)·xs)·(Λ v xs. match_Pair·v·(Λ x y. succeed·(LCons·x·(firsts·xs)))))
    = LCons·x·(firsts·xs)
```

Here the simp rules for match_LNil and match_LCons with match_LCons are applicable, so we rewrite the goal accordingly and then beta-reduce again:

```
goal (1 subgoal):
 1. run·(fail +++ match_Pair·(x, y)·(Λ x y. succeed·(LCons·x·(firsts·xs))))
    = LCons·x·(firsts·xs)
```

Now the simplifier can use the rule for match_Pair applied to a pair constructor. After rewriting and beta-reducing again, we have the following.

```
goal (1 subgoal):
 1. run·(fail +++ succeed·(LCons·x·(firsts·xs))) = LCons·x·(firsts·xs)
```

At this point, the rewrite rules for run and mplus given in Fig. 3.11 are sufficient to solve the goal. All equations in the original specification to Fixrec are proved in a similar manner: Unfold once on the left-hand side, and then simplify. When all of the equations have been proved, the list of theorems is bound to the name firsts.simps; Fixrec also adds all of them to the simplifier.

The fixrec_simp method performs the same proof steps that Fixrec uses internally to prove equations: Substitute the appropriate unfold rule, and then apply the simplifier. The implementation contains some extra machinery to help it find the appropriate unfold rule to use. Using the theory data mechanism [WW07], Fixrec maintains a table that maps from names of constants to their unfold rules. The fixrec_simp method examines the current subgoal to find the name of the leading constant on the left-hand side of the equation, and then looks up the unfold rule from the table.

3.5.6 Mutual recursion

Fixrec can define two or more values at once, where the definition of each one may refer to any of the others—this situation is called *mutual recursion*. We will use an example definition of three mutually recursive infinite lists to show how Fixrec handles this internally.

```
fixrec listA :: "tr llist" and listB :: "tr llist" and listC :: "tr llist"
  where "listA = LCons·TT·listB"
  | "listB = LCons·FF·listC"
  | "listC = LCons·FF·listA"
```

In order to keep the focus on the mutual recursion, this example purposefully does not use pattern matching. (Pattern-match compilation works no differently in the mutually recursive case.)

Fixrec starts to treat mutual recursion specially when it creates the functional to use with the fixed point combinator. In the single-definition case, the pattern match compiler produces a single equation of the form constant = rhs; the right-hand side is then abstracted over all occurrences of the constant. With mutual recursion we now have a *list* of equations, each with a different constant on the left-hand side. Fixrec combines all of the right-hand sides into a tuple, and then abstracts over a tuple of all the constants at once. (A lambda abstraction with a tuple pattern is represented in Isabelle with possibly-nested applications of the constant prod_case :: ('a ⇒ 'b ⇒ 'c) ⇒ 'a × 'b ⇒ 'c.) The result is a functional that maps from tuples to tuples. Fixrec attempts to prove that this functional is continuous, using the tactics described above in Sec. 3.5.4. (Note that cont2cont includes continuity lemmas for both prod_case and Pair.)

```
have listA_listB_listC.cont:
  "cont (λ(a, b, c). (LCons·TT·b, LCons·FF·c, LCons·FF·a))"
```

The least fixed point of this functional is a tuple. Each of the new constants is defined by projecting (using fst and snd) the appropriate component of the tuple.

```
definition listA_def:
  "listA ≡ fst
    (fix·(Abs_cfun (λ(a, b, c). (LCons·TT·b, LCons·FF·c, LCons·FF·a))))"

definition listB_def:
  "listB ≡ fst (snd
    (fix·(Abs_cfun (λ(a, b, c). (LCons·TT·b, LCons·FF·c, LCons·FF·a)))))"

definition listC_def:
  "listC ≡ snd (snd
    (fix·(Abs_cfun (λ(a, b, c). (LCons·TT·b, LCons·FF·c, LCons·FF·a)))))"
```

Now we need to use these definitions together with the continuity lemma to generate unfolding rules and an induction rules. We will consider the unfolding rules first. In the single-definition case, we used the library lemma def_cont_fix_eq to generate the unfolding rule.

```
lemma def_cont_fix_eq:
  "⟦f ≡ fix·(Abs_cfun F); cont F⟧ ⟹ f = F f"
```

Unfortunately, none of the constant definitions have the right form to work with this rule. To remedy this, Fixrec uses another library lemma to combine all of the individual constant definitions into a new theorem that has the right form.

```
lemma Pair_equalI: "⟦x ≡ fst p; y ≡ snd p⟧ ⟹ (x, y) ≡ p"

have listA_listB_listC_def:
  "(listA, listB, listC) ≡
    fix·(Abs_cfun (λ(a, b, c). (LCons·TT·b, LCons·FF·c, LCons·FF·a)))"
```

Now def_cont_fix_eq can be resolved with listA_listB_listC_def and listA_listB_listC.cont to produce the following rule:

```
have listA_listB_listC.unfold_raw:
  "(listA, listB, listC) =
    (λ(a, b, c). (LCons·TT·b, LCons·FF·c, LCons·FF·a)) (listA, listB, listC)"
```

After rewriting to reduce the applications of prod_case, we get the combined unfolding rule:

```
have listA_listB_listC.unfold:
  "(listA, listB, listC) = (LCons·TT·listB, LCons·FF·listC, LCons·FF·listA)"
```

One step remains to produce the individual unfold rules for each constant. Fixrec uses two more library lemmas to go from equalities between tuples to equalities between their components:

```
lemma Pair_eqD1: "(a, b) = (c, d) ⟹ a = c"

lemma Pair_eqD2: "(a, b) = (c, d) ⟹ b = d"

theorem listA.unfold: "listA = LCons·TT·listB"

theorem listB.unfold: "listB = LCons·FF·listC"

theorem listC.unfold: "listC = LCons·FF·listA"
```

This takes care of the unfolding rules. Fixrec goes on to use these for proving the defining equations, as described above in Sec. 3.5.5 (a trivial process in this particular example, because it does not use any pattern matching).

Now we will consider the induction rule that Fixrec produces for mutually recursive definitions. In the single-definition case, simply resolving def_cont_fix_ind with the fixed point definition and continuity lemma yields a suitable induction rule. With mutually recursive definitions, however, getting a usable induction rule will take some more work. When def_cont_fix_ind is resolved with listA_listB_listC_def and listA_listB_listC.cont, we get the following rule:

```
have listA_listB_listC.induct_raw:
  "⟦adm P; P ⊥;
   ⋀x. P x ⟹ P ((λ(a, b, c). (LCons·TT·b, LCons·FF·c, LCons·FF·a)) x)⟧
   ⟹ P (listA, listB, listC)"
```

The serious problem with this rule is in the form of the conclusion. In practice, mutual induction is often used with conjunctions of propositions, such as $\mathsf{Q}(\mathsf{listA}) \wedge \mathsf{R}(\mathsf{listB}) \wedge \mathsf{S}(\mathsf{listC})$ (where Q, R, and S are placeholders for arbitrary predicates). But the raw induct rule above, with its conclusion $\mathsf{P}\ (\mathsf{listA}, \mathsf{listB}, \mathsf{listC})$, does not match goals of this form!

To remedy this problem, Fixrec instantiates the predicate P in the raw induction rule with a predicate that uses prod_case. In our example, P would be instantiated to $(\lambda(\mathsf{a}, \mathsf{b}, \mathsf{c}).\ \mathsf{P}\ \mathsf{a}\ \mathsf{b}\ \mathsf{c})$. After rewriting to simplify applications of prod_case to pairs or $\bot$, and to split universal quantifiers over product types, we get the final mutual induction rule below.

```
theorem listA_listB_listC.induct:
  "⟦adm (λ(a, b, c). P a b c); P ⊥ ⊥ ⊥;
   ⋀a b c. P a b c ⟹ P (LCons·TT·b) (LCons·FF·c) (LCons·FF·a)⟧
   ⟹ P listA listB listC"
```

3.6 DISCUSSION

The current implementation of Fixrec improves on the original version by Amber Telfer [Tel04] in various ways. Internal changes to pattern match compilation have already been discussed in Sec. 3.4. In terms of user-visible features, the original package supported a subset of definitions allowed by the current package: Only one function could be defined at a time (no mutual recursion), and functions were required to have exactly one argument (which could be a tuple). Strict constructor functions (whose match functions have conditional rewrite rules) were not supported. Figure 3.14 summarizes the main differences between the two versions of Fixrec.

	Original	Current
Number of function arguments	exactly 1	0 or more
Mutual recursion	No	Yes
HOL constructors like Pair	No	Yes
Patterns with strict constructors	No	with $\mathsf{x} \neq \bot$
Overlapping patterns	No	with (unchecked)
Proving extra equations	fixpat	fixrec_simp

Figure 3.14: Differences between original (2004) and new versions of FIXREC

Another difference between the old and new versions of FIXREC is the method for proving additional equations. The original FIXREC package provided a top-level command called fixpat, which would be given a list of function patterns. For each pattern, the tool would unfold the function definition once and simplify; the result of the simplification would become the right-hand-side of the generated theorem. In contrast, the new fixrec_simp method requires users to specify a complete equation. Using fixrec_simp is thus a bit more verbose than fixpat, but it has the advantage of being more predictable (because the produced theorems are stated explicitly). It also enables users to prove less trivial equations, because they can be stated with side conditions (like $\mathsf{x} \neq \bot$), and fixrec_simp can be combined with other proof methods such as case analysis on other variables.

The other definition package most closely related to FIXREC is the partial function package for Isabelle/HOL implemented by Krauss [Kra10b] (still in development at time of writing). This tool will use the same approach to pattern match compilation as the FUNCTION package, but instead of well-founded recursion, it uses a domain-theoretic least fixed point combinator to define recursive functions. Instead of a pcpo type class, it uses of a collection of explicit ordering relations for various supported types, each of which is proved to be a complete partial order. For example, a flat ordering is defined for the 'a option type, where None is considered to be below Some x for all x. A similar ordering is provided for an exception monad, where one of the exception values is considered as the bottom element.

Future work. The FIXREC package has several limitations that could be addressed in future work. One such limitation has to do with definedness side conditions: Sometimes FIXREC requires side conditions when it seems they should not be necessary. For example, due to the conditional simplification rule for match_sinl (Fig. 3.13), the following definition currently requires the side condition $x \neq \bot$.

```
fixrec from_sinl :: "'a ⊕ 'b → 'a"
  where "x ≠ ⊥ ⟹ from_sinl·(sinl·x) = x"
```

However, the function from_sinl actually satisfies the equation from_sinl·(sinl·x) = x unconditionally, which can be proven by case analysis on whether or not $x = \bot$. It would be preferable if FIXREC could perform this case analysis automatically, so that users could write this definition without the side condition. One way to accomplish this would be to replace the rewrite rule for match_sinl in Fig. 3.13 with the following rule:

```
lemma match_sinl_sinl [simp]:
  "match_sinl·(sinl·x)·k = (if x = ⊥ then ⊥ else k·x)"
```

The simplifier would then automatically perform a case split on the if-then-else. Such rules would be trivial to prove for sinl, sinr, and spair. However, generating similar rules for constructors defined by the DOMAIN package would require rewriting ML code, and has not been implemented.

Another problem, mentioned already in Sec. 3.5.4, is that the result of pattern match compilation is visible to users as part of the induction rules. It would be preferable to hide such implementation details from the users. As stated before, merely defining the functional as a constant would not be sufficient if users had to unfold its definition to prove anything about it. A more useful solution would be for Fixrec to generate equations for the functional based on the ones provided for the original function. For example, when defining the function firsts, Fixrec could generate a constant firsts_functional with the following theorems:

```
fixrec firsts :: "('a × 'b) llist → 'a llist"
  where firsts_LNil: "firsts·LNil = LNil"
  | firsts_LCons: "firsts·(LCons·(x, y)·xs) = LCons·x·(firsts·xs)"

theorem firsts_functional.simps:
  "firsts_functional r·LNil = LNil"
  "firsts_functional r·(LCons·(x, y)·xs) = LCons·x·(r·xs)"
```

An alternative approach would be to define a nice output syntax for nested case expressions in HOLCF, and use it for compiled patterns; then there would be no pressing need to hide it from users. But this is also a challenge to design, because it would need to work smoothly with simplification. It is desirable to be able to partially simplify a nested case expression, and partially-simplified case expressions must also be pretty-printed in a presentable fashion.

Even with its limitations, the Fixrec package is already quite useful in practice. In Chapter 7 we present some case studies using Fixrec to formalize various Haskell library functions. In another previously published case study, the present author used Fixrec and fixed point induction to verify a substantial real-world Haskell library [Huf09b].

Chapter 4

RECURSIVE DATATYPE DEFINITIONS: THE DOMAIN PACKAGE

4.1 INTRODUCTION

Datatype definitions are a standard feature of many functional programming languages. A datatype definition is a way for a user to specify a new type, by explicitly enumerating all the ways that values of that type may be constructed. A datatype definition, then, consists of a list of *constructors*, each of which may take zero or more arguments of specified types.

For example, in Haskell we might define a datatype `OptInts`, consisting of optional pairs of integers; values of type `OptInts` include `None`, as well as pairs like `Pair 3 5`. (And because we can write non-terminating expressions in Haskell, type `OptInts` actually includes the special value $\perp$ as well.)

```
data OptInts = None | Pair Int Int
```

Datatypes may also be *recursive*, where the type being defined appears on the right-hand side of the definition. Below is an example of a binary tree datatype, where the `Branch` constructor takes two other subtrees as arguments. Values of type `BinTree` include `Tip`, `Branch Tip Tip`, and `Branch Tip (Branch Tip Tip)`; `Branch` constructors may be nested to any depth.

```
data BinTree = Tip | Branch BinTree BinTree
```

Datatypes can also have *type parameters*, such as this type of lists where the element type is specified by the parameter `a`. Values include `Cons 3 (Cons 5 Nil)`, which has type `List Int`.

```
data List a = Nil | Cons a (List a)
```

Isabelle/HOL Datatype package. The DATATYPE package implements these kinds of recursive datatype definitions for Isabelle/HOL. It provides an input syntax that looks very much like the Haskell definitions.

```
datatype opt_ints = None | Pair "int" "int"

datatype bintree = Tip | Branch "bintree" "bintree"

datatype 'a list = Nil | Cons "'a" "'a list"
```

However, unlike Haskell, the DATATYPE package lives and works in a world of inductive data and total functions. Types defined by the DATATYPE package include precisely those values that can be built up using finite combinations of constructor functions; nothing more, nothing less. In particular, they do not include $\perp$, or any infinite values.

Isabelle/HOLCF Domain package. The DOMAIN package provides a similar datatype facility for HOLCF, in a world of cpos, bottoms, infinite values, and continuous functions. Like the DATATYPE package, it also supports a syntax that looks very much like Haskell:

```
domain opt_ints = None | Pair "int lift" "int lift"

domain bintree = Tip | Branch "bintree" "bintree"

domain 'a list = Nil | Cons "'a" "'a list"
```

But unlike the DATATYPE package, the DOMAIN package produces types that are pointed cpos: The constructors are continuous functions, and $\perp$ is an element of every datatype. Types defined by the DOMAIN package may or may not include partial and infinite values, depending on whether the user decides to make the datatype lazy. (Laziness is the default for datatypes in Haskell, but not for strict functional languages like ML.) For example, the lazy version of bintree shown below includes both finite and infinite values, including an infinite value consisting of nothing but Branch constructors all the way down.

domain bintree = Tip | Branch (**lazy** "bintree") (**lazy** "bintree")

The HOLCF Domain package was originally created by David von Oheimb in 1997 [Ohe97], after which it became an integral part of HOLCF '99 [MNOS99]. Since the initial version, the code has undergone a fairly significant amount of modification and improvement by the present author. The main purpose of this chapter is to document the current state of the implementation, and explain the ideas behind all the new code that was not present in the original.

The original Domain package relied on a rather dubious implementation technique: Instead of following the definitional approach, and actually constructing recursive datatypes, it relied on axioms to help define new datatypes. Specifically, each new datatype required three new axioms to be declared; further definitions and proofs were then based on these. In all the years since HOLCF '99, the Domain package has continued to use axioms to support its definitions, although many of the intervening changes to the code have been motivated by the goal of eliminating the reliance on axioms.

At last, in HOLCF '11 the Domain package now has two instantiations—an axiomatic mode (kept for backward compatibility) and a new definitional mode. This chapter describes the operation of the axiomatic version. However, only a small fraction of the code is specific to the axiomatic mode, so most of this chapter is equally applicable to both versions. Relative to the axiomatic mode, the definitional mode requires a significant amount of extra theoretical machinery to implement, which will be built up in the course of Chapters 5 and 6.

Contributions. Much of the material presented in this chapter is just a reworking of von Oheimb's original Domain package implementation [Ohe97]. Since the HOLCF '99 version, many of the present author's improvements to the package have been incremental. However there are a few new features that stand out:

- New modular code organization, to isolate the axiom-generating components
- Efficient method for proving exhaustiveness of constructors, using rewriting
- Notion of *decisive* take functions for recognizing finite-valued domains
- Support for indirect-recursive domain definitions
- Integration with the Fixrec package
- Numerous speed-ups, making large datatype definitions feasible

Overview. The remainder of this chapter consists of two main parts. First, we describe the Domain package from a user's point of view, explaining the various kinds of domain specifications it is possible to write, and the relevant constants and theorems the package generates (§4.2). Next we cover the implementation of the Domain package, showing how it is organized into modules, and how each module works (§4.3). The chapter concludes with a short summary and discussion of the problems caused by axioms, and previews the eventual solution to these problems (§4.4).

4.2 DOMAIN PACKAGE FEATURES

4.2.1 Strict and lazy constructors

The Domain package has an input syntax similar to the Isabelle/HOL Datatype package [NPW02]. The right-hand side of a domain definition consists of one or more constructors, each with zero or more argument types.

```
domain 'a strictlist = nil | cons "'a" "'a strictlist"
```

Each domain definition produces constructors with continuous function types. For example, defining 'a strictlist as shown yields the function cons with type 'a → 'a strictlist → 'a strictlist. Constructor functions are strict by default, so cons·⊥·s = ⊥ and cons·a·⊥ = ⊥. Constructors can be made non-strict in specified arguments using the **lazy** keyword.

```
domain 'a stream = SNil | SCons "'a" (lazy "'a stream")
```

Note that making the recursive argument lazy causes type 'a stream to include infinite values, in constrast with 'a strictlist, whose values are all finite.

The Domain package also generates a collection of rewrite rules about the constructors, which are added to the simplifier.

```
"SCons·⊥·s = ⊥"
"SNil ≠ ⊥"
"SCons·a·s = ⊥ ⟷ a = ⊥"
"a ≠ ⊥ ⟹ SCons·a·s ⊑ SCons·a'·s' ⟷ a ⊑ a' ∧ s ⊑ s'"
"a ≠ ⊥ ⟹ SCons·a·s = SCons·a'·s' ⟷ a = a' ∧ s = s'"
"SNil ⋢ SCons·a'·s'"
"SCons·a·s ⊑ SNil ⟷ a = ⊥"
"SNil ≠ SCons·a'·s'"
"SCons·a·s ≠ SNil"
"compact SNil"
"⟦compact a; compact s⟧ ⟹ compact (SCons·a·s)"
```

In addition to the simplification rules, the Domain package also generates theorems asserting that the constructors are exhaustive. The generated theorems follow a naming scheme similar to the Isabelle/HOL Datatype package, where each theorem is qualified by the name of the relevant type. The two logically equivalent theorems below follow the same pattern as the similarly-named theorems generated by Datatype, except that they also include cases for ⊥ and strict constructor functions.

```
theorem stream.nchotomy:
  "y = ⊥ ∨ y = SNil ∨ (∃a s. y = SCons·a·s ∧ a ≠ ⊥)"
```

```
theorem stream.exhaust:
  "⟦y = ⊥ ⟹ P; y = SNil ⟹ P; ⋀a s. ⟦y = SCons·a·s; a ≠ ⊥⟧ ⟹ P⟧ ⟹ P"
```

The Domain package registers stream.exhaust as the default case analysis rule for type 'a stream. This means proof methods like **apply** (cases y) can be used on a stream variable y, without having to explicitly name rule stream.exhaust.

4.2.2 Case expressions

The Domain package configures Isabelle's parser to allow case expressions on each new datatype. This case syntax is supported by *case combinators.* For example, with the 'a stream domain, the case expression case x of SNil ⇒ y | SCons·a·s ⇒ z translates to stream_case·y·(Λ a s. z)·x. The case combinator stream_case, which has type 'b → ('a → 'a stream → 'b) → 'a stream → 'b, satisfies the following simplification rules:

```
theorem stream.case_rews [simp]:
  "stream_case·f1·f2·⊥ = ⊥"
  "stream_case·f1·f2·SNil = f1"
  "a ≠ ⊥ ⟹ stream_case·f1·f2·(SCons·a·s) = f2·a·s"
```

4.2.3 Mixfix syntax

In Isabelle, a *mixfix* declaration specifies custom syntax for parsing and pretty printing; infix syntax is a special case for functions of two arguments. The Domain package supports mixfix declarations for data constructors. For example, we can specify syntax for SNil and an infix operator for the SCons constructor, like this:

```
domain 'a stream =
  SNil ("<>") | SCons "'a" (lazy "'a stream") (infixr "##" 60)
```

Now <> is defined as alternative syntax for SNil, and a ## s abbreviates SCons·a·s.

Mixfix declarations can also be given for the type constructors themselves. For example:

```
domain ('a, 'b) either (infixl ":+:" 20) = Left (lazy "'a") | Right (lazy "'b")
```

This introduces the type notation 'a :+: 'b as an abbreviation for ('a, 'b) either.

4.2.4 Selector functions

Each constructor argument in a domain declaration may be given an optional selector name. For example, for our 'a stream datatype we can label the arguments of SCons as head and tail:

```
domain 'a stream = SNil | SCons (head :: "'a") (lazy tail :: "'a stream")
```

The Domain package then defines two selector functions: head :: 'a stream → 'a and tail :: 'a stream → 'a stream. When applied to the correct constructor, each selector function projects out the desired argument; applied to any other constructor, the selector returns ⊥. Accordingly, the package derives the following rewrite rules, which are added to the simplifier.

```
theorem stream.sel_rews [simp]:
  "head·⊥ = ⊥"
  "tail·⊥ = ⊥"
  "head·SNil = ⊥"
  "tail·SNil = ⊥"
  "head·(SCons·a·s) = a"
  "a ≠ ⊥ ⟹ tail·(SCons·a·s) = s"
```

4.2.5 Discriminator functions

The Domain package automatically defines a discriminator function for each data constructor. Each discriminator function returns a lifted boolean, depending on its argument: TT if its argument is the correct constructor, FF for a different constructor, and ⊥ if its argument is ⊥. For example, when defining type 'a stream, the package generates is_SNil :: 'a stream → tr and is_SCons :: 'a stream → tr. These functions satisfy the following rules, which are declared to the simplifier.

```
theorem stream.dis_rews [simp]:
  "is_SNil·⊥ = ⊥"
  "is_SCons·⊥ = ⊥"
  "is_SNil·x = ⊥ ⟷ x = ⊥"
```

```
"is_SCons·x = ⊥ ⟷ x = ⊥"
"is_SNil·SNil = TT"
"a ≠ ⊥ ⟹ is_SNil·(SCons·a·s) = FF"
"is_SCons·SNil = FF"
"a ≠ ⊥ ⟹ is_SCons·(SCons·a·s) = TT"
```

4.2.6 Fixrec package support

The Domain package generates a match combinator for each data constructor, and registers them for use with the Fixrec package. Most of the details are irrelevant as far as users are concerned; the important thing is that after defining a datatype with the Domain package, users can write function definitions over that datatype using Fixrec.

4.2.7 Take functions

Each datatype defined by the Domain package gets its own *take function*, which is actually a chain of functions that return finite approximations of their input values. These take functions are a bit like the Haskell function `take` for lists, in that take (Suc n) applied to a constructor maps take n over that constructor's recursive arguments. However, unlike the Haskell `take` function, take 0 is undefined (i.e., $\bot$). In this way, the HOLCF take functions are exactly like the generic *approx* functions discussed by Hutton and Gibbons [HG01].

For the 'a stream datatype, the Domain package defines a function stream_take whose type is nat ⇒ 'a stream → 'a stream. The defining equations below are added as default simplification rules.

```
"stream_take 0 = ⊥"
"stream_take n·⊥ = ⊥"
"stream_take (Suc n)·SNil = SNil"
"stream_take (Suc n)·(SCons·a·s) = SCons·a·(stream_take n·s)"
```

The package also derives a few more theorems from the definition of stream_take, which can also be useful for simplification:

```
theorem stream.chain_take [simp]: "chain (λn. stream_take n)"

theorem stream.take_below: "stream_take n·x ⊑ x"

theorem stream.take_take:
  "stream_take m·(stream_take n·x) = stream_take (min m n)·x"
```

In practice, take functions are usually used for low-level induction proofs. The Domain package generates a *reach axiom* (in two equivalent forms) stating that the least upper bound of the chain of take functions is the identity. The principle of *take induction* (rule stream.take_induct) is a direct corollary of the reach axiom.

```
theorem stream.lub_take: "(⨆n. stream_take n) = ID"

theorem stream.reach: "(⨆n. stream_take n·x) = x"

theorem stream.take_induct: "⟦adm P; ⋀n. P (stream_take n·x)⟧ ⟹ P x"
```

The *take lemma* is another reasoning principle derived from the reach axiom. This lets users show that two (possibly infinite) values are equal, by showing that the finite values returned by the take functions are equal.

```
theorem stream.take_lemma:
  "(⋀n. stream_take n·x = stream_take n·y) ⟹ x = y"
```

A variation of the take lemma, specific to lazy lists, was originally popularized by Bird and Wadler [BW88]. The take lemma generated by the Domain package is actually identical to the generic "approximation lemma" described by Hutton and Gibbons [HG01]. An example of a proof using the take lemma can be found in the case study in Chapter 7.

4.2.8 Induction rules

In addition to the low-level take induction rules, the Domain package also generates some high-level induction principles in terms of the constructor functions.

```
theorem stream.finite_induct:
  "⟦P ⊥; P SNil; ⋀a s. ⟦a ≠ ⊥; P s⟧ ⟹ P (SCons·a·s)⟧
    ⟹ P (stream_take n·x)"
```

To be able to conclude that a property holds for *all* streams, including infinite ones, the full induction rule adds an admissibility requirement.

```
theorem stream.induct:
  "⟦adm P; P ⊥; P SNil; ⋀a s. ⟦a ≠ ⊥; P s⟧ ⟹ P (SCons·a·s)⟧ ⟹ P x"
```

For mutually recursive domain definitions, the DOMAIN package generates a mutual induction rule for proving properties of both types simultaneously.

```
domain 'a list1 = Nil1 | List2 (lazy "'a list2")
  and 'a list2 = Cons2 (lazy "'a") (lazy "'a list1")
```

```
theorem list1_list2.induct:
  "⟦adm P1; adm P2; P1 ⊥; P1 Nil1; ⋀list2. P2 list2 ⟹ P1 (List2·list2);
    P2 ⊥; ⋀a list1. P1 list1 ⟹ P2 (Cons2·a·list1)⟧ ⟹ P1 x1 ∧ P2 x2"
```

The DOMAIN package declares the high-level induction rules as the default induction rules for their types. This means that, for example, the proof method **apply** (induct s) will use rule stream.induct if s is a variable of type 'a stream. Likewise, the simultaneous induction method **apply** (induct x **and** y) will automatically use rule list1_list2.induct if x and y have the appropriate types.

4.2.9 Finite-valued domains

Some datatypes, like 'a strictlist below, contain only finite values because recursion only occurs via strict constructor arguments. For such datatypes, the DOMAIN package can generate induction rules without an admissibility condition. This includes both the low-level take induction principle, and the ordinary high-level induction rule.

```
domain 'a strictlist = nil | cons "'a" "'a strictlist"
```

```
theorem strictlist.take_induct: "(⋀n. P (strictlist_take n·x)) ⟹ P x"
theorem strictlist.induct:
  "⟦P ⊥; P nil; ⋀a s. ⟦a ≠ ⊥; s ≠ ⊥; P s⟧ ⟹ P (cons·a·s)⟧ ⟹ P x"
```

The DOMAIN package can also recognize finite-valued domains in mutually recursive definitions.

4.2.10 Coinduction

The DOMAIN package implements the principle of *coinduction* for recursive domains, following the design described by Pitts [Pit94]. The main concept underlying coinduction is the *bisimulation* relation: A binary relation R is a bisimulation if any two values related by R are either both ⊥, or else they are the same constructor, with their arguments again related by R.

```
definition stream_bisim :: "('a stream ⇒ 'a stream ⇒ bool) ⇒ bool"
  where "stream_bisim R ≡ (∀x y. R x y ⟶
    (x = ⊥ ∧ y = ⊥) ∨
    (x = SNil ∧ y = SNil) ∨
    (∃a s a' s'. a = a' ∧ R s s' ∧ x = SCons·a·s ∧ y = SCons·a'·s'))"
```

The principle of coinduction states that any two values related by any bisimulation relation must be equal.

```
theorem stream.coinduct: "⟦stream_bisim R; R x y⟧ ⟹ x = y"
```

A good exposition of the proof technique of coinduction, as used for lazy lists, can be found in Gibbons and Hutton [GH05]. The implementation of coinduction for recursive domains has changed little since the HOLCF '99 DOMAIN package, and we will not be using coinduction further in this dissertation.

4.2.11 Indirect recursion

In all the recursive domain definitions shown so far, occurrences of recursive types have only appeared directly as constructor arguments. But it is also possible for a recursive type to occur *indirectly*, under one or more other type constructors. For

```
definition prod_map :: "('a → 'b) → ('c → 'd) → 'a × 'c → 'b × 'd"
  where "prod_map = (Λ f g (x, y). (f·x, g·y))"

definition sprod_map :: "('a → 'b) → ('c → 'd) → 'a ⊗ 'c → 'b ⊗ 'd"
  where "sprod_map = (Λ f g. (:x y:). (:f·x, g·y:)))"

definition ssum_map :: "('a → 'b) → ('c → 'd) → 'a ⊕ 'c → 'b ⊕ 'd"
  where "ssum_map = (Λ f g. sscase·(sinl oo f)·(sinr oo g))"

definition u_map :: "('a → 'b) → 'a u → 'b u"
  where "u_map = (Λ f. fup·(up oo f))"

definition cfun_map :: "('b → 'a) → ('c → 'd) → ('a → 'c) → ('b → 'd)"
  where "cfun_map = (Λ a b f x. b·(f·(a·x)))"
```

Figure 4.1: Map combinators for various HOLCF types

example, in the following definition of bintree, the argument type of the Branch constructor contains recursive occurrences of bintree within a strict product.

```
domain bintree = Tip | Branch "bintree ⊗ bintree"
```

Unlike the HOLCF '99 Domain package, the HOLCF '11 version now supports such indirect-recursive domain definitions. One user-visible consequence of indirect recursion is that the rewrite rules for take functions mention *map combinators*:

```
"bintree_take (Suc n)·(Branch·x) =
  Branch·(sprod_map·(bintree_take n)·(bintree_take n)·x)"
```

Here sprod_map is a combinator that maps each of two functions over the respective elements of a strict pair. There are similar map combinators for a few other type constructors that the Domain package also knows about; these are shown in Fig. 4.1. In Chapter 6 when we talk about the definitional Domain package, we will see how to make this list extensible, in a sound way.

For indirect-recursive domains, the Domain package still generates take induction rules, just as it does for any other domain definition. However, currently it does not produce high-level induction or coinduction rules. Formulating high-level induction rules for arbitrary indirect-recursive domains is still an experimental topic; we will have more to say about this in the case study in Chapter 7.

4.3 IMPLEMENTATION

The entire implementation of the Domain package is based on pairs of functions that form *domain isomorphisms*. Each isomorphism relates the new "abstract" type, such as 'a stream, to a "representation" type built from strict sums and products, plus lifting to model laziness. For example, consider our lazy stream type:

domain 'a stream = SNil | SCons (head :: "'a") (**lazy** tail :: "'a stream")

Here 'a stream is the abstract type and one $\oplus$ ('a $\otimes$ 'a $\text{stream}_\perp$) is the representation type. The Domain package produces continuous abs and rep functions between these two types, which are analogous to the Abs and Rep functions produced by the Typedef package described in Chapter 2.

stream_abs :: "one $\oplus$ ('a $\otimes$ 'a $\text{stream}_\perp$) $\to$ 'a stream"

stream_rep :: "'a stream $\to$ one $\oplus$ ('a $\otimes$ 'a $\text{stream}_\perp$)"

Together with standard operations on the strict sum, strict product, and lifting types, the abs and rep functions can be used to define all the necessary operations on new domain types—such as constructors, case combinators, and take functions. Most of the properties of these operations are derived from the *isomorphism axioms*, which assert that the abs and rep functions are each other's inverses.

Since HOLCF '99, the Domain package has undergone a significant reorganization to make the code more modular. The diagram in Fig. 4.2 shows the main components, and how information is passed between them. The new organization offers multiple benefits. For example, all code dealing with constructor functions is now isolated in components in the bottom half of the diagram. The other modules,

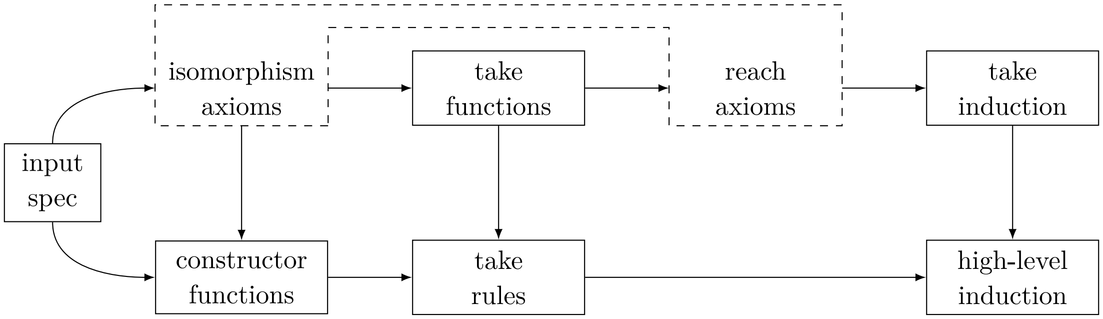


Figure 4.2: Domain package implementation schematic

located in the top half, only need to know about domain isomorphisms, and never see any information about constructor functions. These modules can then have simpler interfaces, and also fewer dependencies on the other components, making maintenance easier.

Another benefit of the new modularization is the ability to swap out individual components. In particular, note that all code involved with generating axioms is contained within the pair of modules inside the dashed lines in Fig. 4.2. With clear interfaces between these axiomatic modules and the rest of the Domain package, it will be possible to replace them with definitional versions at a later time, with minimal modifications to the rest of the system. We will come back to the definitional version of the Domain package in Chapter 6.

Each module in Fig. 4.2 will be explained in more detail in one of the following sections. For each module, we will describe the definitions and proofs that are generated, as well as the ML record types that make up the interfaces.

4.3.1 Input specification module

The input-specification module of the Domain package starts by parsing its input according to the grammar shown in Fig. 4.3. The input consists of one or more domain specifications, each of which has a type on the left-hand side, and a

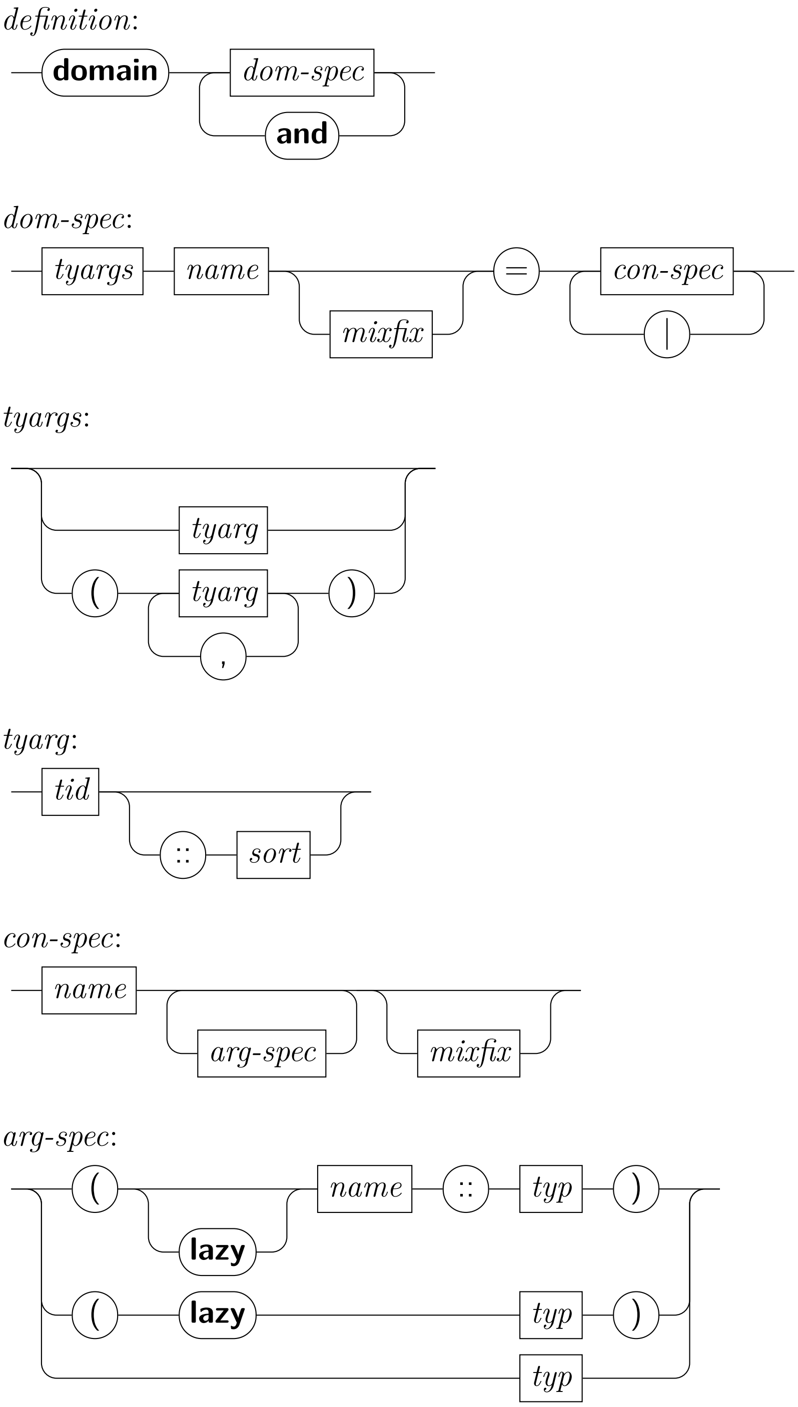


Figure 4.3: Input syntax for Domain package

list of constructor specifications on the right. Each constructor has zero or more arguments, either lazy or strict, with an optional selector name for each.

After parsing, the module performs some simple checks, like making sure that there are no duplicate type or constructor names. There are also some more semantic checks: For example, the type of every strict constructor argument must be in class pcpo. (Class cpo is sufficient for lazy arguments that do not have a selector function.) Another test ensures that recursive occurrences of types are used with the same type arguments. Indirect recursion is also checked for, ensuring that it is only used with acceptable type constructors, including strict sums and products, lifting, and continuous function space. (Recursion under the full function space type constructor is specifically *not* allowed, because it can lead to unsound isomorphism axioms.)

After performing these basic checks on the input, the main task of this module is to prepare information to be passed into the next modules. The input specification module passes information to two others: the isomorphism axioms module and the constructor functions module.

For the isomorphism axioms module, the input specification is transformed into a list of domain equations, one for each new type; information about constructors and selectors is discarded. For example, when defining **domain** 'a stream = SNil | SCons "'a" (**lazy** "'a stream"), the isomorphism module will only see the domain equation $\mathsf{'a\ stream} \cong \mathsf{one} \oplus (\mathsf{'a} \otimes \mathsf{'a\ stream}_{\perp})$. Mutually recursive domain definitions yield multiple domain equations, one for each type. The ML type passed to the isomorphism axioms module is `(binding * mixfix * (typ * typ)) list`, which comprises a type name (ML type `binding`), type constructor syntax (ML type `mixfix`), and the domain equation for each type.

For the constructor functions module, information about the constructors is assembled, using ML type `(binding * (bool * binding option * typ) list *`

```
type iso_info =
  {
    absT        : typ,     repT        : typ,
    abs_const   : term,    rep_const   : term,
    abs_inverse : thm,     rep_inverse : thm
  }
```

Figure 4.4: Record type for domain isomorphisms

mixfix) list. For each constructor, we have its name, the list of argument specifications, and its syntax; each argument has a boolean laziness flag, an optional selector name, and the argument type. With mutually recursive domain definitions, such a list is assembled for each new type.

4.3.2 Isomorphism axioms module

The input to the isomorphism axioms module consists of a name, syntax, and a domain equation for each new type. The module performs the following steps: First, it *declares* each new type constructor—instead of *defining* them with Typedef or Cpodef, they are simply declared without a definition. Second, a pcpo class instance is axiomatically asserted for each type. Third, a pair of abs and rep functions corresponding to each domain equation is declared—again, without providing a definition. Fourth, isomorphism axioms are generated, asserting that each pair of abs and rep functions is an isomorphism.

The types, constants, and axioms generated by the isomorphism axioms module are collected in the iso_info record type, shown in Fig. 4.4. The module produces one such record for each mutually recursive domain.

The supporting theory files for the Domain package define a binary predicate called iso, which asserts that its two arguments form a continuous isomorphism.

```
definition iso :: "('a → 'b) ⇒ ('b → 'a) ⇒ bool"
  where "iso abs rep ⟷ (∀x. rep·(abs·x) = x) ∧ (∀y. abs·(rep·y) = y)"
```

```
type take_info =
  {
    take_consts          : term list,   take_defs     : thm list,
    chain_take_thms      : thm list,    take_0_thms   : thm list,
    take_strict_thms     : thm list,    take_Suc_thms : thm list,
    deflation_take_thms  : thm list
  }
```

Figure 4.5: Record type for take functions and related theorems

The axioms declared by the isomorphism module are sufficient to derive iso abs rep as a theorem. (An alternative design would be to declare iso abs rep directly as the axiom.) Other modules will use the iso predicate to derive various other properties of the abs and rep functions.

4.3.3 Take functions module

The take functions module accepts input of type `(binding * iso_info) list`, comprising a type name and isomorphism information for each new type. The module defines a take function for each type and generates several theorems about them, which are collected in an ML record of type `take_info` (Fig. 4.5).

Defining the take functions. Below is the take function for the stream datatype, which is a solution to $\text{'a stream} \cong \text{one} \oplus (\text{'a} \otimes \text{'a stream}_{\perp})$. The take function stream_take n is defined as the nth iteration of a certain functional, consisting of stream_abs and stream_rep composed with a combination of map functions. The map functions are in a one-to-one correspondence with the type constructors that appear on the right-hand side of the domain equation.

```
definition stream_take :: "nat ⇒ 'a stream → 'a stream"
  where "stream_take n ≡ iterate n·(Λ f.
    stream_abs oo ssum_map·ID·(sprod_map·ID·(u_map·f)) oo stream_rep)·⊥"
```

```
lemma [domain_map_ID]:
  "u_map·(ID :: 'a → 'a) = (ID :: 'a⊥ → 'a⊥)"
  "prod_map·(ID :: 'a → 'a)·(ID :: 'b → 'b) = (ID :: 'a × 'b → 'a × 'b)"
  "sprod_map·(ID :: 'a → 'a)·(ID :: 'b → 'b) = (ID :: 'a ⊗ 'b → 'a ⊗ 'b)"
  "ssum_map·(ID :: 'a → 'a)·(ID :: 'b → 'b) = (ID :: 'a ⊕ 'b → 'a ⊕ 'b)"
  "cfun_map·(ID :: 'a → 'a)·(ID :: 'b → 'b) = (ID :: ('a → 'b) → ('a → 'b))"
```

Figure 4.6: Extensible set of rules with the domain_map_ID attribute

To keep track of the map combinators associated with each type constructor, the Domain package maintains a database of theorems with the [domain_map_ID] attribute. Using a theorem attribute makes it relatively easy to add support for additional type constructors, by adding new theorems to the database. The theorems making up the initial contents of the database are shown in Fig. 4.6. We can use these rules to generate a combination of map functions corresponding to any given complex type expression: Starting with the identity function ID at the given type, perform rewriting with the domain_map_ID rules, applying them right-to-left.

For mutually recursive domains, the take functions are defined in a manner similar to how Fixrec handles mutual recursion: We construct a functional that operates on tuples, and then project out the desired component using fst and snd.

```
domain 'a list1 = Nil1 | List2 "'a list2" and 'a list2 = Cons2 "'a" "'a list1"

definition list1_take :: "nat ⇒ 'a list1 → 'a list1"
  where "list1_take n ≡ fst (iterate n·
        (Λ f. (list1_abs oo ssum_map·ID·(snd f) oo list1_rep,
           list2_abs oo sprod_map·ID·(fst f) oo list2_rep))·⊥)"

definition list2_take :: "nat ⇒ 'a list2 → 'a list2"
  where "list2_take n ≡ snd (iterate n·
        (Λ f. (list1_abs oo ssum_map·ID·(snd f) oo list1_rep,
           list2_abs oo sprod_map·ID·(fst f) oo list2_rep))·⊥)"
```

Note that unlike mutual recursion, indirect recursion does not require any special treatment. Of the following two variations of the bintree datatype, one is indirect-recursive and the other is not. However, they both give rise to the same domain equation, and so the take function is defined identically for either one.

```
domain bintree = Tip | Branch "bintree ⊗ bintree"

domain bintree = Tip | Branch "bintree" "bintree"
```

Proving theorems about take functions. After defining the take functions, the take function module proves several theorems about them. We will consider the stream datatype as an example.

```
theorem stream.chain_take [simp]: "chain (λn. stream_take n)"

theorem stream.take_0 [simp]: "stream_take 0 = ⊥"

theorem stream.take_Suc: "stream_take (Suc n) = stream_abs oo
  ssum_map·ID·(sprod_map·ID·(u_map·(stream_take n))) oo stream_rep"
```

These first few lemmas are easy to prove. Theorem stream.chain_take follows from the fact that (λn. iterate n·F·⊥) is always a chain. (For mutually recursive take functions, note that fst and snd are monotone, and thus preserve chains.) The take_0 and take_Suc theorems follow directly from the definitions by the properties of iterate.

To help derive the other properties of stream_take, we make use of a new concept: A *deflation* is a continuous function f that is idempotent and below the identity function, so that f oo f = f ⊑ ID. These are properties that are expected to hold for every take function. We define the deflation predicate as follows.[1]

```
definition deflation :: "('a → 'a) ⇒ bool"
  where "deflation d ⟷ (∀x. d·(d·x) = d·x) ∧ (∀x. d·x ⊑ x)"
```

[1] The deflation predicate is actually defined with a **locale** command, which generates a definition equivalent to the one shown here.

```
lemma [domain_deflation]:
  "deflation ID"
  "deflation f ⟹ deflation (u_map·f)"
  "⟦deflation f; deflation g⟧ ⟹ deflation (prod_map·f·g)"
  "⟦deflation f; deflation g⟧ ⟹ deflation (sprod_map·f·g)"
  "⟦deflation f; deflation g⟧ ⟹ deflation (ssum_map·f·g)"
  "⟦deflation f; deflation g⟧ ⟹ deflation (cfun_map·f·g)"
```

Figure 4.7: Extensible set of rules with the domain_deflation attribute

Showing that each take function is a deflation is the only really non-trivial proof done by the take functions module. Once it is proved that stream_take is a chain of deflations, the module can easily derive the rest of the desired theorems about stream_take.

```
theorem stream.deflation_take: "deflation (stream_take n)"
```

The proof of stream.deflation_take proceeds by induction on n. After unfolding stream.take_0, the base case is trivial, because ⊥ is a deflation. In the inductive case, we start by unfolding stream.take_Suc. A suitably-instantiated rule deflation_abs_rep is applied next, followed by deflation lemmas for the various map combinators. The Domain package maintains a database of deflation lemmas that have the [domain_deflation] attribute; Fig. 4.7 shows the initial set of domain_deflation rules.

```
lemma deflation_abs_rep:
  "⟦iso abs rep; deflation d⟧ ⟹ deflation (abs oo d oo rep)"
```

For mutually recursive domains, the deflation_take rules for all take functions must be proved simultaneously, with a single induction on n. The other steps in the deflation proof are similar to the single-domain case.

After proving stream.deflation_take, we can derive the other theorems. Rules stream.take_below and stream.take_strict follow directly from stream.deflation_take.

Theorem stream.take_take also depends on the fact that stream_take is a chain, using the library lemma deflation_chain_min.

```
theorem stream.take_below: "stream_take n·x ⊑ x"

theorem stream.take_strict: "stream_take n·⊥ = ⊥"

theorem stream.take_take:
  "stream_take m·(stream_take n·x) = stream_take (min m n)·x"

lemma deflation_chain_min:
  "⟦chain t; ⋀n. deflation (t n)⟧ ⟹ t m·(t n·x) = t (min m n)·x"
```

Finally, the take functions module collects the relevant theorems in a `take_info` record, to be passed along to the later modules.

4.3.4 Reach axioms module

For each take function, the Domain package declares an axiom stating that the least upper bound of the chain of take functions is the identity function. For example, with the 'a stream type, the following axiom is declared:

```
axioms stream.lub_take: "(⨆n. stream_take n) = ID"
```

For mutually recursive domain definitions, one axiom is declared for each new type. This list of axioms is then passed along to the take induction module.

4.3.5 Take induction module

The take induction module takes three pieces of input: a list of type `(binding * iso_info) list` from the isomorphism axioms module, a `take_info` record from the take functions module, and a `thm list` containing lub_take theorems from the reach axioms module. The take induction module then derives a few low-level induction principles involving the take functions, and packages up the results in an ML record of type `take_induct_info` (Fig. 4.8).

```
type take_induct_info =
  {
    take_info          : take_info,
    lub_take_thms      : thm list,
    reach_thms         : thm list,
    take_lemma_thms    : thm list,
    is_finite          : bool,
    take_induct_thms   : thm list
  }
```

Figure 4.8: Record type for DOMAIN package theorems related to take induction

For each new type, the take induction module generates three new theorems in addition to the lub_take axiom: the reach lemma, the take lemma, and the take induction rule. For the stream datatype, these are as follows.

```
theorem stream.reach: "(⨆n. stream_take n·x) = x"

theorem stream.take_lemma:
  "(⋀n. stream_take n·x = stream_take n·y) ⟹ x = y"

theorem stream.take_induct: "⟦adm P; ⋀n. P (stream_take n·x)⟧ ⟹ P x"
```

Each of these is derived easily from theorems stream.chain_take and stream.lub_take, using an appropriate library lemma.

```
lemma lub_ID_reach:
  assumes "chain t" and "(⨆n. t n) = ID"
  shows "(⨆n. t n·x) = x"

lemma lub_ID_take_lemma:
  assumes "chain t" and "(⨆n. t n) = ID"
  shows "(⋀n. t n·x = t n·y) ⟹ x = y"

lemma lub_ID_take_induct:
  assumes "chain t" and "(⨆n. t n) = ID"
  shows "⟦adm P; ⋀n. P (t n·x)⟧ ⟹ P x"
```

Finite-valued domains. The take induction module's job is not always quite this easy. Recall that strict recursive datatypes contain only finite values: In these cases we must generate a take induction rule without an admissibility condition.

To determine whether or not a domain (or set of mutually recursive domains) is finite-valued, the take induction module performs a test on its associated domain equation. For example, consider the strict list datatype:

```
domain 'a strictlist = nil | cons "'a" "'a strictlist"
```

The domain equation is 'a strictlist $\cong$ one $\oplus$ ('a $\otimes$ 'a strictlist). Note that the only type constructors surrounding the recursive occurrence of 'a strictlist on the right-hand side are the strict sum and strict product. This indicates that 'a strictlist is a finite-valued domain. Equivalently, we might notice that the only map combinators mentioned in the definition of strictlist_take are ssum_map and sprod_map.

```
theorem strictlist.take_Suc: "strictlist_take (Suc n) = strictlist_abs oo
  ssum_map·ID·(sprod_map·ID·(strictlist_take n)) oo strictlist_rep"
```

On the other hand, for lazy streams we have 'a stream $\cong$ one $\oplus$ ('a $\otimes$ 'a $\text{stream}_\perp$) as the domain equation. The presence of lifting on the recursive occurrence of 'a stream indicates that this is *not* a finite-valued domain.

In order to prove a take induction principle for 'a strictlist without an admissibility condition, we will show that strictlist_take satisfies a particular property, which we call *decisiveness*. (Note that decisiveness is not a standard concept in domain theory, but rather an invention of the present author.) A decisive function is a particular kind of deflation that makes an all-or-nothing choice for each input value: The function either returns its input value, or else it returns $\perp$. The HOLCF definition of the predicate decisive is shown in Fig. 4.9, along with several related lemmas. Note that decisiveness is preserved by ssum_map and sprod_map, but not by any of the other map combinators; this is why it is important that only ssum_map and sprod_map are used in the definition of strictlist_take.

```
definition decisive :: "('a → 'a) ⇒ bool"
  where "decisive f ⟷ (∀x. f·x = x ∨ f·x = ⊥)"

lemma decisive_bottom: "decisive ⊥"

lemma decisive_ID: "decisive ID"

lemma decisive_ssum_map: "⟦decisive f; decisive g⟧ ⟹ decisive (ssum_map·f·g)"

lemma decisive_sprod_map: "⟦decisive f; decisive g⟧ ⟹ decisive (sprod_map·f·g)"

lemma decisive_abs_rep: "⟦iso abs rep; decisive f⟧ ⟹ decisive (abs oo f oo rep)"
```

Figure 4.9: Definition and properties of decisive deflations

The proof of decisive (strictlist_take n) proceeds by induction on n. Each case is solved with the help of the various lemmas about decisive shown in Fig. 4.9. The proof structure is exactly the same as for the deflation_take proofs done by the take functions module.

Having proved the decisiveness of strictlist_take, the take induction rule can be derived using the library lemma lub_ID_finite_take_induct.

```
lemma lub_ID_finite_take_induct:
  assumes "chain t" and "(⨆n. t n) = ID" and "⋀n. decisive (t n)"
  shows "(⋀n. P (t n·x)) ⟹ P x"
```

To prove this lemma, it is sufficient to show that there exists n such that t n·x = x. Now consider the chain (λn. t n·x), which has a finite range: Because each t n is decisive, the chain's range is a subset of {⊥, x}. As a finite chain, it must attain its least upper bound at some point.

Ultimately the take induction module wraps up all the new theorems in a `take_induct_info` record to pass along to the induction rules module. This record also includes the original `take_info` record, and a boolean flag to indicate finite-valued domains.

```
type constr_info =
  {
    iso_info   : iso_info,
    con_specs : (term * (bool * typ) list) list,
    con_betas : thm list,
    nchotomy   : thm,          exhaust    : thm,
    compacts   : thm list,     con_rews   : thm list,
    inverts    : thm list,     injects    : thm list,
    dist_les   : thm list,     dist_eqs   : thm list,
    case_rews : thm list,     sel_rews   : thm list,
    dis_rews   : thm list,     match_rews : thm list
  }
```

Figure 4.10: Record type for constructor-related constants and theorems

4.3.6 Constructor functions module

The constructor functions module is called with three pieces of input: a `binding` for the type name, an `iso_info` record from the isomorphism axioms module, and a list of constructor specifications of type `(binding * (bool * binding option * typ) list * mixfix) list`. Each call to the module deals with a single domain equation; with mutually recursive definitions the module is called multiple times, once for each new type.

The constructor functions module returns a `constr_info` record filled with generated constants and theorems (Fig. 4.10). The remainder of this section is organized into subheadings, each corresponding to one or more fields of this record.

The 'a stream type is a bit too simple to adequately demonstrate all the features of the constructor functions module. Instead, we will use this slightly more complex ('a, 'b) tree datatype as a running example.

```
domain ('a, 'b) tree =
  Tip | Leaf "'a" |
  Node (left :: "('a, 'b) tree") (lazy middle :: "'b") (right :: "('a, 'b) tree")
```

Defining the constructors. Each constructor is defined using the tree_abs function from the isomorphism together with some combination of the HOLCF data constructors spair, sinl, sinr, ONE, and up.

```
"Tip = abs_tree·(sinl·ONE)"
"Leaf = (Λ a. abs_tree·(sinr·(sinl·a)))"
"Node = (Λ t1 b t2. abs_tree·(sinr·(sinr·(:t1, up·b, t2:))))"
```

At the time the constructor constants are defined, the module also declares any infix syntax that may have been specified.

After defining the constructor functions, the module fills in the `con_specs` field of result record with an ML value of type `(term * (bool * typ) list) list`. For each constructor, we have the actual constructor constant (ML type `term`) and a list of argument specifications, each consisting of a laziness flag and an argument type. We discard information about syntax and selector functions, because the induction rules module does not need it.

After defining the constructor functions, the next step is to prove rules for unfolding the definitions of fully-applied constructors. These rules occupy the `con_betas` field of the results record, and will be used in many other internal proofs.

Each constructor function is defined with continuous lambda abstractions, which require continuity checks to beta-reduce. Domain definitions with many-argument constructors produce equally large numbers of continuity conditions, which can get expensive to check. For this reason, we use the bottom-up continuity proof method described in Chapter 2, which saves time by reusing common subproofs.

```
have con_betas:
  "Tip = abs_tree·(sinl·ONE)"
  "⋀a. Leaf·a = abs_tree·(sinr·(sinl·a))"
  "⋀ t1 b t2. Node·t1·b·t2 = abs_tree·(sinr·(sinr·(:t1, up·b, t2:)))"
```

The original HOLCF '99 Domain package did not have an equivalent of the con_betas rules. Instead, every proof about the constructors simply unfolded the raw constructor definitions, and beta-reduced them using the simplifier. Thus the same continuity checks were done again and again, separately in each generated theorem—and even worse, the time required for each check was exponential in the number of nested lambdas. For domain definitions with constructors of more than two or three arguments, the continuity proofs dominated the entire running time of the Domain package, and constructors with more than four or five arguments were simply infeasible. In contrast, the HOLCF '11 Domain package scales much better to large numbers of constructor arguments; continuity checks are no longer a performance bottleneck.

Exhaustiveness of constructors. After defining the constructors, the Domain package proceeds to prove that the constructors are exhaustive. The HOLCF '11 Domain package uses a novel proof method, based on rewriting, to generate the exhaustiveness theorem efficiently; we will examine it in some detail.

The method involves generating a type-specific exhaustiveness rule starting from a generic one, using a set of type-directed rewrite rules. The generic starting rule, which is valid for any pcpo, is called exh_start; it merely states that any value p either is or is not equal to ⊥.

```
lemma exh_start: "p = ⊥ ∨ (∃x. p = x ∧ x ≠ ⊥)"
```

Next, we have the set of type-directed rewrite rules:

```
lemma ex_bottom_iffs:
  "(∃x. P x ∧ x ≠ ⊥) ⟷ (∃x. P (sinl·x) ∧ x ≠ ⊥) ∨ (∃x. P (sinr·x) ∧ x ≠ ⊥)"
  "(∃x. P x ∧ x ≠ ⊥) ⟷ (∃x y. (P (:x, y:) ∧ x ≠ ⊥) ∧ y ≠ ⊥)"
  "(∃x. P x ∧ x ≠ ⊥) ⟷ (∃x y. P (:up·x, y:) ∧ y ≠ ⊥)"
  "(∃x. P x ∧ x ≠ ⊥) ⟷ (∃x. P (up·x))"
  "(∃x. P x ∧ x ≠ ⊥) ⟷ P ONE"
```

Each of the rewrite rules has the same pattern on the left-hand side, namely $(\exists x.\ P\ x \wedge x \neq \bot)$. However, each of these rules places a different type constraint on the variable x. Also note that many of the rules' right-hand sides also contain instances of the same pattern, but at smaller types. So rewriting with ex_bottom_iffs may continue for several steps, but it always terminates.

Now we will step through the process of deriving an exhaustiveness theorem for the ('a, 'b) tree datatype. We start by instantiating the theorem exh_start at a type similar to the representation type for ('a, 'b) tree.

$$\textbf{have}\ \mathsf{thm1}\text{: "}(p :: \mathsf{one} \oplus {}'a \oplus ({}'c \otimes {}'b_\bot \otimes {}'d)) = \bot \vee (\exists x.\ p = x \wedge x \neq \bot)\text{"}$$

Next, we rewrite this intermediate theorem, using the rules in ex_bottom_iffs.

$$\begin{aligned}&\textbf{have}\ \mathsf{thm2}\text{: "}p = \bot \vee p = \mathsf{sinl}\cdot\mathsf{ONE} \vee (\exists x.\ p = \mathsf{sinr}\cdot(\mathsf{sinl}\cdot x) \wedge x \neq \bot) \vee \\ &\quad (\exists x\ y\ z.\ (p = \mathsf{sinr}\cdot(\mathsf{sinr}\cdot(:x,\ \mathsf{up}\cdot y,\ z:)) \wedge x \neq \bot) \wedge z \neq \bot)\text{"}\end{aligned}$$

Then thm2 is rewritten to re-associate the conjunctions to the right, resulting in another temporary theorem thm3. Now we will use thm3 to prove that the constructors of domain ('a, 'b) tree are exhaustive:

$$\begin{aligned}&\textbf{theorem}\ \mathsf{tree.nchotomy}\text{:}\\ &\quad \text{"}y = \bot \vee y = \mathsf{Tip} \vee (\exists a.\ y = \mathsf{Leaf}\cdot a \wedge a \neq \bot) \vee \\ &\qquad (\exists t1\ b\ t2.\ y = \mathsf{Node}\cdot t1\cdot b\cdot t2 \wedge t1 \neq \bot \wedge t2 \neq \bot)\text{"}\end{aligned}$$

The proof starts by unfolding con_betas. Next, we can rewrite occurrences of $x = \mathsf{tree\_abs}\cdot y$ to $\mathsf{tree\_rep}\cdot x = y$, and $x = \bot$ to $\mathsf{tree\_rep}\cdot x = \bot$, using rules derived from the isomorphism axioms. This produces a goal that can be solved directly by the temporary theorem thm3.

After proving tree.nchotomy, the case analysis rule tree.exhaust can be derived from it.

$$\begin{aligned}&\textbf{theorem}\ \mathsf{tree.exhaust}\text{:}\\ &\quad \text{"}[\![y = \bot \Longrightarrow P;\ y = \mathsf{Tip} \Longrightarrow P;\ \textstyle\bigwedge a.\ [\![y = \mathsf{Leaf}\cdot a;\ a \neq \bot]\!] \Longrightarrow P;\\ &\qquad \textstyle\bigwedge t1\ b\ t2.\ [\![y = \mathsf{Node}\cdot t1\cdot b\cdot t2;\ t1 \neq \bot;\ t2 \neq \bot]\!] \Longrightarrow P]\!] \Longrightarrow P\text{"}\end{aligned}$$

The derivation starts by composing tree.nchotomy with exh_casedist0; this rule transforms a theorem with conclusion R into an elimination rule of the form (R ⟹ P) ⟹ P. This rule is then rewritten with exh_casedists, yielding theorem tree.exhaust.

```
lemma exh_casedist0: "⟦R; R ⟹ P⟧ ⟹ P"

lemma exh_casedists:
  "((P ∨ Q ⟹ R) ⟹ S) ≡ (⟦P ⟹ R; Q ⟹ R⟧ ⟹ S)"
  "((∃x. P x) ⟹ Q) ≡ (⋀x. P x ⟹ Q)"
  "(P ∧ Q ⟹ R) ≡ (⟦P; Q⟧ ⟹ R)"
```

These proof methods for generating nchotomy and exhaust theorems are far faster than the methods used by the HOLCF '99 DOMAIN package. In the earlier version, these theorems were proved by repeatedly performing case analyses on strict sum and product types. Many redundant cases involving $\bot$ were produced, each of which had to be solved by the simplifier. The number of such steps was proportional to the size of the definition; the time spent on this one proof made up a significant portion of the running time for the DOMAIN package. In contrast, the new version calls a short, finite list of proof tactics that all run quickly.

Simplification rules for constructors. The constructor functions module generates several groups of simplification rules about the constructors: There are rules about compactness, strictness, definedness, distinctness, injectivity, and order comparisons.

- tree.compacts: For each constructor, we get a rule stating that the constructor is compact if all its arguments are compact.

  ```
  "compact Tip"
  "compact a ⟹ compact (Leaf·a)"
  "⟦compact t1; compact b; compact t2⟧ ⟹ compact (Node·t1·b·t2)"
  ```

- tree.con_rews (1): For each non-lazy argument position of each constructor, we get a strictness rule showing that the constructor applied to ⊥ equals ⊥.

```
"Leaf·⊥ = ⊥"
"Node·⊥·b·t2 = ⊥"
"Node·t1·b·⊥ = ⊥"
```

- tree.con_rews (2): For each constructor, we get a definedness rule stating that the constructor equals ⊥ if and only if one of its non-lazy arguments equals ⊥. The proofs are by unfolding con_betas and simplifying with definedness rules for tree_abs, sinl, sinr, spair, up, and ONE.

```
"Tip ≠ ⊥"
"Leaf·a = ⊥ ⟷ a = ⊥"
"Node·t1·b·t2 = ⊥ ⟷ t1 = ⊥ ∨ t2 = ⊥"
```

- tree.dist_les: For each pair of distinct constructors, we get a rule stating that constructor 1 is below constructor 2 if and only if one of the non-lazy arguments of constructor 1 is ⊥.

```
"Tip ⋢ Leaf·a'"
"Tip ⋢ Node·t1'·b'·t2'"
"Leaf·a ⊑ Tip ⟷ a = ⊥"
"Leaf·a ⊑ Node·t1'·b'·t2' ⟷ a = ⊥"
"Node·t1·b·t2 ⊑ Tip ⟷ t1 = ⊥ ∨ t2 = ⊥"
"Node·t1·b·t2 ⊑ Leaf·a' ⟷ t1 = ⊥ ∨ t2 = ⊥"
```

- tree.dist_eqs: For each pair of distinct constructors, we get a rule stating that constructor 1 is equal to constructor 2 if and only if both constructors have at least one non-lazy argument equal to ⊥.

```
"Tip ≠ Leaf·a'"
"Tip ≠ Node·t1'·b'·t2'"
"Leaf·a ≠ Tip"
"Leaf·a = Node·t1'·b'·t2' ⟷ a = ⊥ ∧ (t1' = ⊥ ∨ t2' = ⊥)"
"Node·t1·b·t2 ≠ Tip"
"Node·t1·b·t2 = Leaf·a' ⟷ (t1 = ⊥ ∨ t2 = ⊥) ∧ a' = ⊥"
```

- tree.inverts: For each constructor (except those with no arguments) we have a rule stating that two applications of the same constructor are related by (⊑) if and only if their arguments are also pointwise related by (⊑); this is under the assumption that none of the strict arguments on the left-hand side are ⊥.

  ```
  "Leaf·a ⊑ Leaf·a' ⟷ a ⊑ a'"
  "⟦t1 ≠ ⊥; t2 ≠ ⊥⟧
    ⟹ Node·t1·b·t2 ⊑ Node·t1'·b'·t2' ⟷ t1 ⊑ t1' ∧ b ⊑ b' ∧ t2 ⊑ t2'"
  ```

- tree.injects: An injectivity rule is generated for each constructor that has at least one argument: Two applications of the same constructor are equal if and only if the corresponding arguments are equal. These rules use the same definedness assumptions as tree.inverts.

  ```
  "Leaf·a = Leaf·a' ⟷ a = a'"
  "⟦t1 ≠ ⊥; t2 ≠ ⊥⟧
    ⟹ Node·t1·b·t2 = Node·t1'·b'·t2' ⟷ t1 = t1' ∧ b = b' ∧ t2 = t2'"
  ```

The proofs for all of these simplification rules are essentially the same. Each starts by unfolding the constructor definitions, using con_betas. The proofs are then completed by calling the simplifier with a particular set of rewrite rules: Because each constructor is defined in terms of the basic HOLCF constructors spair, sinl, sinr, and up, the various properties (compactness, strictness, injectivity, etc.) of the new constructors derive from similar properties of these basic constructors.

In addition to spair, sinl, sinr, and up, the constructor definitions also mention abs functions, like tree_abs for the ('a, 'b) tree datatype. Accordingly, rules about compactness, strictness, injectivity, etc. for tree_abs are also needed to complete the proofs. Each of these properties of tree_abs can be derived from the isomorphism axioms, using a few library lemmas.

```
lemma iso.compact_abs: "⟦iso abs rep; compact x⟧ ⟹ compact (abs·x)"
```

```
lemma iso.abs_strict: "iso abs rep ⟹ abs·⊥ = ⊥"

lemma iso.abs_bottom_iff: "iso abs rep ⟹ abs·x = ⊥ ⟷ x = ⊥"

lemma iso.abs_below: "iso abs rep ⟹ abs·x ⊑ abs·y ⟷ x ⊑ y"

lemma iso.abs_eq: "iso abs rep ⟹ abs·x = abs·y ⟷ x = y"
```

Most of the same simplification rules for constructors were also generated by the original HOLCF '99 Domain package, but not necessarily in the same form. For example, con_rews used to have conditional rules like $a \neq \bot \Longrightarrow \mathsf{SCons}\cdot a\cdot s \neq \bot$; the if-and-only-if formulation preferred by the HOLCF '11 version works better as a simplification rule.

Case combinator. The case combinator tree_case is defined in terms of the case combinators for the lifted unit, strict sum, strict product, and lifted cpo types, together with tree_rep.

```
sscase :: ('a → 'c) → ('b → 'c) → 'a ⊕ 'b → 'c
ssplit :: ('a → 'b → 'c) → 'a ⊗ 'b → 'c
fup :: ('a → 'b) → 'a⊥ → 'b
one_case :: 'a → one → 'a
```

Recall from Chapter 2 that ssplit, fup, and one_case all have special syntax as continuous lambda abstractions with patterns.

The tree_case function takes arguments f1, f2, and f3—one corresponding to each constructor. For each of these, a lambda abstraction of a strict tuple (using ssplit) is built, with one element for each constructor argument. Each lazy argument position has an additional up pattern (using fup). For zero-argument constructors, a ONE pattern (using one_case) takes the place of a strict tuple pattern. All of these abstractions are then combined with sscase.

```
definition tree_case :: "'c → ('a → 'c)
   → (('a, 'b) tree → 'b → ('a, 'b) tree → 'c) → ('a, 'b) tree → 'c"
```

```
  where "tree_case ≡ (Λ f1 f2 f3. sscase·(Λ ONE. f1)·
    (sscase·(Λ a. f2·a)·(Λ (:t1, up·b, t2:). f3·t1·b·t2)) oo tree_rep)"
```

Simplification rules are generated for the case combinator applied to ⊥, and to each constructor.

```
lemma tree.case_rews [simp]:
  "tree_case·f1·f2·f3·⊥ = ⊥"
  "tree_case·f1·f2·f3·Tip = f1"
  "a ≠ ⊥ ⟹ tree_case·f1·f2·f3·(Leaf·a) = f2·a
  "⟦t1 ≠ ⊥; t2 ≠ ⊥⟧ ⟹ tree_case·f1·f2·f3·(Node·t1·b·t2) = f3·t1·b·t2"
```

The proofs are by unfolding the definition of tree_case, unfolding con_betas, and then calling the simplifier. To speed up the beta-reduction of the nested lambdas in the definition of tree_case, the constructor functions module uses the same bottom-up continuity prover used to generate con_betas; this helps to avoid performance problems in domain definitions with large numbers of constructors.

Selector functions. A selector function is produced for each constructor argument that has been given a selector name. Each selector is defined as a composition of the rep function with some sequence of functions from this list:

```
sfst :: 'a ⊗ 'b → 'a
ssnd :: 'a ⊗ 'b → 'b
sscase·ID·⊥ :: 'a ⊕ 'b → 'a
sscase·⊥·ID :: 'a ⊕ 'b → 'b
fup·ID :: 'a⊥ → 'a
```

For example, below are the definitions of the left and middle selectors, which project arguments of the Node constructor of the ('a, 'b) tree datatype.

```
definition left :: "('a, 'b) tree → ('a, 'b) tree"
  where "left ≡ sfst oo sscase·⊥·ID oo sscase·⊥·ID oo tree_rep"

definition middle :: "('a, 'b) tree → 'b"
  where "middle ≡
    fup·ID oo sfst oo ssnd oo sscase·⊥·ID oo sscase·⊥·ID oo tree_rep"
```

The constructor functions module produces rules for each selector applied to $\bot$, and to each constructor. Definedness conditions are only needed when the selector is applied to the correct constructor, and then only for strict arguments *other than* the one being selected.

```
lemma tree.sel_rews [simp]:
  "left·⊥ = ⊥"
  "left·Tip = ⊥"
  "left·(Leaf·a) = ⊥"
  "t2 ≠ ⊥ ⟹ left·(Node·t1·b·t2) = t1"
```

Only the rules for the selector left are shown here; similar rules are generated for the selectors middle and right. They are proved by unfolding the selector definitions, unfolding con_betas, and calling the simplifier.

Discriminator functions. For each constructor, a discriminator function is defined in terms of the case combinator. One branch returns TT, and the rest return FF. Below is the definition of the discriminator is_Leaf for the ('a, 'b) tree datatype, in terms of tree_case.

```
definition is_Leaf :: "('a, 'b) tree → tr"
  where "is_Leaf ≡ tree_case·FF·(Λ a. TT)·(Λ t1 b t2. FF)"
```

Discriminators for the Tip and Node constructors are defined similarly. For each discriminator, we generate rules for the discriminator applied to each constructor as well as to $\bot$. These rules follow directly from the rewrite rules for tree_case. We also generate if-and-only-if definedness rules like $\mathsf{is\_Leaf}\cdot\mathsf{x} = \bot \longleftrightarrow \mathsf{x} = \bot$, which are proved by case analysis on x.

Fixrec match combinators. As with the discriminator functions, we also define the match combinators for Fixrec in terms of the case combinator tree_case. Each match combinator takes two arguments: the scrutinee x, and the match continuation k. The continuation is used for the case branch of the matching constructor; every other branch returns fail, indicating pattern match failure.

```
definition match_Tip :: "('a, 'b) tree → 'c match → 'c match"
  where "match_Tip ≡ (Λ x k. tree_case·k·(Λ a. fail)·(Λ t1 b t2. fail)·x)"

definition match_Leaf :: "('a, 'b) tree → ('a → 'c match) → 'c match"
  where "match_Leaf ≡ (Λ x k. tree_case·fail·k·(Λ t1 b t2. fail)·x)"

definition match_Node ::
    "('a, 'b) tree → (('a, 'b) tree → 'b → ('a, 'b) tree → 'c match) → 'c match"
  where "match_Node ≡ (Λ x k. tree_case·fail·(Λ a. fail)·k·x)"
```

For each match combinator, we generate rules for applications to each constructor as well as to ⊥. As with the discriminator rules, these are proved by simplification with tree.case_rews.

```
lemma tree.match_rews [simp]:
  "match_Leaf·⊥·k = ⊥"
  "match_Leaf·Tip·k = fail"
  "a ≠ ⊥ ⟹ match_Leaf·(Leaf·a)·k = k·a"
  "⟦t1 ≠ ⊥; t2 ≠ ⊥⟧ ⟹ match_Leaf·(Node·t1·b·t2)·k = fail"
```

Only the rules for match_Leaf are shown here; in actuality tree.match_rews also includes similar rules for match_Tip and match_Node.

In addition to proving simplification rules for the match combinators, the constructor functions module also registers each match combinator with the Fixrec package, associating them with the corresponding constructors.

4.3.7 Take rules module

This is a small module whose sole purpose is to produce simplification rules for the take functions applied to constructors. As input, it gets a `take_info` record from the take functions module, and a list of `constr_info` records from the constructor functions module. It produces lists of take_rews theorems, which are passed on to the induction rules module.

```
theorem tree.take_rews [simp]:
  "tree_take (Suc n)·Tip = Tip"
  "tree_take (Suc n)·(Leaf·a) = Leaf·a"
  "tree_take (Suc n)·(Node·t1·b·t2) = Node·(tree_take n·t1)·b·(tree_take n·t2)"
```

Note that because take functions are strict, definedness assumptions are not needed on any of the take_rews theorems. The proofs proceed by unfolding con_betas and tree.take_Suc, and then calling the simplifier.

4.3.8 Induction rules module

As input, the induction rules module gets a `take_induct_info` record from the take induction module, and a list of `constr_info` records from the constructor functions module; it also receives lists of take_rews theorems from the take rules module. In turn, it produces high-level induction rules. (It also defines a bisimulation predicate and proves a coinduction rule. But because this part of the Domain package has changed little since the original version [Ohe97, MNOS99], and coinduction is not used elsewhere in this thesis, we omit a full description of its implementation.)

As a first step toward proving the high-level induction rule, the induction rules module starts by proving the *finite* induction rule. It assumes that each constructor preserves some predicate P; the conclusion asserts that P must then hold for any output of the take function.

```
theorem tree.finite_induct:
  assumes "P ⊥" and "P Tip" and "⋀a. a ≠ ⊥ ⟹ P (Leaf·a)"
    and "⋀t1 b t2. ⟦t1 ≠ ⊥; t2 ≠ ⊥; P t1; P t2⟧ ⟹ P (Node·t1·b·t2)"
  shows "P (tree_take n·x)"
```

To prove the finite induction rule, the first step is to use the assumption "P ⊥" to derive stronger, unconditional versions of the other assumptions. These can be shown by case analysis on whether each argument equals ⊥, using the strictness rules for the constructors.

```
have "P Tip" and "⋀a. P (Leaf·a)"
  and "⋀t1 b t2. ⟦P t1; P t2⟧ ⟹ P (Node·t1·b·t2)"
```

Using these strengthened assumptions, we proceed to show ∀x. P (tree_take n·x) by induction on n. For the base case n = 0, the goal simplifies to P ⊥, which matches one of the assumptions. In the n = Suc n' case, we do a case analysis on x, and then simplify each subcase with tree.take_rews. Each subgoal is then discharged using the strengthened assumptions together with the inductive hypothesis.

The main induction rule makes the exact same assumptions about P as the finite induction rule, but has a more general conclusion. It is derived directly from tree.finite_induct by composing it with the take induction rule tree.take_induct.

```
theorem tree.take_induct: "(⋀n. P (tree_take n·x)) ⟹ P x"
```

```
theorem tree.induct:
  assumes "P ⊥" and "P Tip" and "⋀a. a ≠ ⊥ ⟹ P (Leaf·a)"
    and "⋀t1 b t2. ⟦t1 ≠ ⊥; t2 ≠ ⊥; P t1; P t2⟧ ⟹ P (Node·t1·b·t2)"
  shows "P x"
```

Note that type ('a, 'b) tree is finite-valued, because all of the recursive constructor arguments are strict. Thus, neither tree.take_induct nor tree.induct requires an admissibility assumption. On the other hand, the lazy 'a stream datatype contains infinite values, and its take induction rule stream.take_induct does have an admissibility assumption. The high-level induction rule stream.induct thus inherits the admissibility requirement from stream.take_induct.

```
theorem stream.induct:
  "⟦adm P; P ⊥; P SNil; ⋀a s. ⟦a ≠ ⊥; P s⟧ ⟹ P (SCons·a·s)⟧ ⟹ P x"
```

The proof scripts used to generate high-level induction rules only work if recursive occurrences of types are used directly as type constructor arguments—they do not work with indirect recursion. For this reason, the induction rules module explicitly tests for indirect recursion; if it is detected, the proofs of the high-level induction rules are skipped.

4.4 DISCUSSION

The Domain package offers HOLCF users an easy way to define new recursive datatypes. In addition to defining datatypes and constructors, the Domain package also generates numerous auxiliary functions and theorems, giving HOLCF users an easy way to *reason* about their new datatypes.

The HOLCF '11 Domain package improves over the original HOLCF '99 version in several ways. Many of the improvements help to expand the set of programs that it is possible to reason about in HOLCF, which is one of the goals set out in Chapter 1. For example, the HOLCF '11 version is faster and scales better to larger definitions, making it possible to verify Haskell programs with large, complex datatypes that were previously infeasible. The support for indirect recursion also expands the universe of datatypes that can be formalized in HOLCF, but in a different direction.

The integration with the Fixrec package is very important by this measure because it lets users formalize programs that do pattern-matching on user-defined datatypes—something found in nearly every Haskell program. Having Fixrec integration means that users can translate more programs directly from Haskell to HOLCF, without having to rearrange them to work around limitations of the theorem prover.

4.4.1 Problems with axioms

The LCF theorem prover architecture, with a small proof kernel that implements an abstract theorem type, is designed to keep the trusted code base to a bare minimum—this is the code that users *have to trust* in order to believe that the system is sound. The prover can then be extended with new definition packages, or other arbitrary code that lives outside the kernel, without increasing the trusted code base at all. Because all theorems are constructed ultimately by kernel operations, the theorems are still guaranteed to be correct.

This soundness argument can break down if users can freely declare axioms. If an inconsistent set of axioms is declared, then it becomes possible to derive false theorems—to have soundness, users must trust that all axioms in the system are consistent. So if a package requires a few new axioms, the correctness of those axioms must be trusted. But if a package *generates* new axioms, then we have to trust *all the code involved in generating the axioms.* A bug in any of this code can cause the prover to become unsound.

This is exactly what happened with the HOLCF '99 Domain package. The consistency of the kinds of axioms that arise from simple recursive datatype definitions was justified by an informal proof on paper [Ohe97]. There was nothing wrong with the informal proof—indeed, as long as the Domain package was used in a manner consistent with the expectations of the designer, the axioms it generated were always consistent.

However, a bug in the implementation allowed the Domain package to accept some definitions that were outside the scope of the informal proof—specifically, indirect-recursive definitions. It turns out that indirect-recursive definitions involving strict sums and products, lifting, or continuous function space are generally sound. However, indirect recursion with the full function space is not sound in general.

domain paradox = MkParadox "paradox ⇒ one"

The HOLCF '99 Domain package would accept this definition of paradox, and axiomatize an isomorphism between types paradox and paradox ⇒ one. Recall that one is a two-element type, so this is essentially an isomorphism between a (nonempty) type and its powerset. A proof of False can be derived from this with a bit more work.

This particular bug has been patched; the HOLCF ’11 Domain package now tests for indirect recursion, and disallows it except for type constructors that it knows are safe. However, the overall situation is not much better than before: As long as axioms are still being generated, it is difficult to trust the soundness of the whole system.

Most of the rest of this dissertation focuses on a solution to this problem: How to implement a trustworthy Domain package that doesn’t take any shortcuts, and uses explicit definitions instead of declaring axioms. But before getting into the semantics of recursive datatypes, the next chapter focuses instead on another language feature: nondeterminism. Along the way, we will develop some infrastructure which will eventually be useful for implementing a definitional Domain package.

Chapter 5

POWERDOMAINS AND IDEAL COMPLETION

5.1 INTRODUCTION

Powerdomains are a domain-theoretic analog of powersets, which were designed for reasoning about the semantics of nondeterministic programs [Plo76]. In turn, nondeterminism can be used to model other features of real-world programs, such as concurrency [Pap01a, Thi95] and exceptions [PJRH^{+}99].

This chapter describes the first fully-mechanized formalization of powerdomains, which was originally presented in earlier work by the present author [Huf08]. It is implemented in the Isabelle theorem prover as part of HOLCF '11. 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.

The development of powerdomains in HOLCF '11 follows the ideal completion construction presented by Gunter and Scott [GS90, §5.2]. Some alternative constructions are also given by Abramsky and Jung [AJ94, §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.

One side-benefit that came from the powerdomain formalization effort is the HOLCF '11 ideal completion library. Originally created specifically to construct powerdomains, it is now generally useful for constructing other cpos in HOLCF, particularly the universal domain (see Chapter 6).

Another significant aspect of the work described in this chapter is the identification of a suitable category of domains—the *bifinite* domains—to serve as the default class in HOLCF '11. This was a forward-looking design decision: The choice was made not just because of the requirements of the powerdomain library, but also considering the eventual requirements of the universal domain and the definitional Domain package (Chapter 6), and the relationships among all of these libraries and tools. The end result is a powerdomain library that integrates seamlessly with the definitional Domain package (see the concurrency case study in Chapter 7).

Contributions. The original contributions presented in this chapter comprise features of the powerdomain library itself, as well as parts of the supporting libraries that are more generally useful.

- Formalization of three powerdomain types (upper, lower, and convex) with which users can reason about nondeterministic programs
- Collections of coordinated rewrite rules that provide automation for solving comparisons between powerdomain values
- A formalization of the category of bifinite domains
- A general library of ideal completion that can be reused for defining other cpo types

Overview. This chapter starts by motivating the definition of powerdomains, pointing out the limitations of powersets and Haskell datatypes for modeling nondeterministic computation (§5.2). Next we examine the three main varieties of powerdomains, and attempt to convey some intuitions about their structures and what each is good for (§ 5.3). For readers wishing to use the HOLCF '11 powerdomain library, we then summarize all of the powerdomain operations provided by the library, as well as some of the lemmas and proof automation that is available

(§5.4). A description of the implementation of the library follows: We explain the general process of ideal completion and its formalization in HOLCF (§5.5), define a class of bifinite cpos that work with it (§5.6), and then show how ideal completion is used to implement the powerdomain library (§5.7). Finally, we have a comparison with previous work and discuss possible applications of the powerdomain library (§5.8).

5.2 NONDETERMINISM MONADS

From a functional programmer's perspective, a powerdomain can be thought of as simply a special kind of monad for nondeterminism. A *monad* is a type constructor that represents computations; different monads can model computations with different kinds of side-effects. Every monad has a *return* operation to represent *pure* computations (i.e., computations with no side-effects) and a *bind* operation to represent sequencing of computations. In addition to return and bind, a powerdomain also provides a binary operation for making a nondeterministic choice; the nondeterminism can be considered as a kind of side-effect of the computation.

In the remainder of this section, we will consider a few different monads that can represent nondeterminic computations. Each example satisfies some, but not all, of the required properties of a powerdomain.

The set monad. One way to model nondeterministic computations is using sets: A nondeterministic computation returning a value of type A can be modeled as a set $S \in \mathcal{P}(A)$ of possible return values. Similarly, a parameterized computation taking input of type A and returning output of type B can be modeled as a function $f : A \to \mathcal{P}(B)$.

We can build up such sets and functions from smaller components using a few basic operations. First, a singleton set like $\{x\}$ represents a pure computation (i.e., one that is completely deterministic). Second, a general union like $\bigcup_{x \in S} f(x)$ represents sequenced computations: The set S models the first computation, whose result is bound to the variable x; then the set $f(x)$ represents the second computation, which may depend on the output of the first. Finally, a binary union like $S \cup T$ represents a nondeterministic choice: Each set models an alternative computation, and the combined computation chooses randomly which branch to take.

These operations make sets into a nondeterminism monad, where $\mathit{return}(x) = \{x\}$ and $\mathit{bind}(S, f) = \bigcup_{x \in S} f(x)$. Binary union serves as the nondeterministic choice operator. The operations of the set monad satisfy several useful laws:

$$\textstyle\bigcup_{x \in \{a\}} f(x) = f(a) \tag{5.1}$$

$$\textstyle\bigcup_{x \in A} \{x\} = A \tag{5.2}$$

$$\textstyle\bigcup_{y \in (\bigcup_{x \in A} f(x))} g(y) = \bigcup_{x \in A} \bigcup_{y \in f(x)} g(y) \tag{5.3}$$

$$\textstyle\bigcup_{x \in (A \cup B)} f(x) = (\bigcup_{x \in A} f(x)) \cup (\bigcup_{x \in B} f(x)) \tag{5.4}$$

$$(A \cup B) \cup C = A \cup (B \cup C) \tag{5.5}$$

$$A \cup B = B \cup A \tag{5.6}$$

$$A \cup A = A \tag{5.7}$$

The first three mention only return and bind. These are called the *monad laws*, and are well known to Haskell programmers: Monad instances in Haskell are generally expected to satisfy them. The remaining four laws are specific to the nondeterministic choice operator. Law (5.4) says that bind distributes over choice, and laws (5.5)–(5.7) state that choice is associative, commutative and idempotent. The operations of any powerdomain must satisfy all seven laws; we will refer to them collectively as the *powerdomain laws*.

Haskell syntax for nondeterminism monads. In order to compare the same computations evaluated in different nondeterminism monads, we will introduce a common syntax for nondeterministic computations using Haskell type classes. Haskell already comes with a standard type class for monads, with return and bind operations:

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

Recall that Haskell provides special syntax to make it easy to write monadic code. The expression `do {x <- m; k}` is syntactic sugar for `m >>= (\x -> k)`, and `do {a; b; c}` is shorthand for `do {a; do {b; c}}`.

On top of the `Monad` class, we can define a subclass for monads with a binary nondeterministic choice operator [PM00]:

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

Now we can write Haskell code for computations that will run in any nondeterminism monad, using the `do` notation and the `|+|` operation. The types of such computations will have the class context `(ChoiceMonad m)`. Figure 5.1 shows the seven powerdomain laws; these are simply Eqs. (5.1)–(5.7) translated into Haskell syntax.

The Haskell list monad. 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.

```
(++)           :: [a] -> [a] -> [a]
[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

```
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
```

Figure 5.1: The powerdomain laws in Haskell syntax

```
instance Monad [] where
  return x        = [x]
  []        >>= f = []
  (x : xs) >>= f = f x ++ (xs >>= f)

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

Compared to the set monad, 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. The list monad also satisfies some equational laws: The return and bind functions satisfy the monad laws (1–3), bind distributes over choice (Law 4), and append is associative (Law 5).

However, the list monad does not satisfy the last two laws—the choice operator for lists is neither commutative nor idempotent. This means that 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 $\perp$ or `undefined`). The lists `[3,5,undefined]` and `[5,5,3,undefined,3]` both represent the value of `f` equally well; both represent the set $\{3, 5, \perp\}$.

The list monad also suffers from the opposite problem: In some circumstances, it identifies computations that should be considered distinct. The difficulty is caused by the append operation for lists, which does not behave well in the presence of infinite output. The problem is that 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.

This problem is demonstrated by the following recursive nondeterministic computations. The possible results of `comp1` include all the positive *even* integers, while *all* integers greater than or equal to 2 are possible results of `comp2`.

```
comp1 :: (ChoiceMonad m) => m Int
comp1 = do {x <- return 0 |+| comp1; return (x+2)}

comp2 :: (ChoiceMonad m) => m Int
comp2 = do {x <- return 0 |+| comp2 |+| return 1; return (x+2)}
```

When interpreted in the list monad, `comp1` returns the infinite list `[2,4,6,8,...]` which correctly includes every possible return value of the computation. We should expect `comp2` to additionally include odd integers, but when we evaluate `comp2` in the list monad, we get exactly the same list as `comp1`—all of the odd integers are missing. The reason is that because the denotation of `comp2` is an infinite list, the "`return 1`" branch is never reached.

The Haskell tree monad. Another possible nondeterminism monad for Haskell is the binary tree, whose definition is shown below.

```
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 ChoiceMonad Tree where
  l |+| r = Node l r
```

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. When we interpret `comp2` in the tree monad, the resulting infinite tree contains all of its possible return values, including both even and odd integers. Thus the tree monad can always distinguish any two computations that have different sets of return values.

However, the problem of multiple representations remains; in fact this problem is even worse than before. The list append operator was at least associative, but because the choice operator for trees is a data constructor, it does not satisfy any non-trivial equalities. The tree monad thus satisfies only the first four of the seven powerdomain laws.

The set monad, revisited. We have seen that the set monad satisfies all seven of the powerdomain laws, unlike the Haskell list or tree monads. However, compared to those Haskell monads, the set monad has a major limitation: It is not always possible to define recursive computations.

In an earlier chapter (§3.3) we saw how to express recursive definitions in terms of a least fixed point combinator. To use a fixed point combinator on the set monad, we need to impose an ordering on the powerset $\mathcal{P}(A)$ that makes it into a pointed cpo. The subset ordering $(\subseteq)$ on $\mathcal{P}(A)$ yields a pointed cpo, and we can show that the bind and union operations are continuous under this ordering. But unless A is a discrete cpo, the return operation $\{-\} : A \to \mathcal{P}(A)$ is not continuous, or even monotone.

As a consequence, the set monad only works with some recursively-defined computations. For example, consider the computation `comp2` that we used previously.

```
comp2 :: (ChoiceMonad m) => m Int
comp2 = do {x <- return 0 |+| comp2 |+| return 1; return (x+2)}
```

When interpreted in the set monad, `comp2` denotes the least fixed point of the function $S \mapsto \bigcup_{x\in(\{0\}\cup S\cup\{1\})}\{x+2\}$. This function is continuous because S occurs only within bind and union operations, which are themselves continuous. The fixed point evaluates (correctly) to the infinite set $\{2,3,4,5,\ldots\}$.

Other definitions simply do not work with the least fixed point operator. For example, consider the following program, which is a computation that itself returns one of two further computations.

```
comp3 :: (ChoiceMonad m) => m (m Int)
comp3 = return (return 1) |+|
        return (do {c <- comp3; x <- c; return (x+2)})
```

The intended meaning of `comp3` is something like $\{\{1\},\{3,5,7,9,\ldots\}\}$. But we cannot model this definition with the fixed point operator, because the recursive call to `comp3` is inside the argument to `return`, which is not continuous in the set monad. Indeed, if we start with `undefined` and evaluate successive unfoldings of `comp3`, we get the sequence of sets $S_0 = \{\}$, $S_1 = \{\{1\},\{\}\}$, $S_2 = \{\{1\},\{3\}\}$, $S_3 = \{\{1\},\{3,5\}\}$, $S_4 = \{\{1\},\{3,5,7\}\}$, $\ldots$ which is not even a chain.

Another limitation of the set monad is that it does not work with recursive datatype definitions. It is often useful to combine monads with datatypes, so that the components of data structures can be computations with side-effects like nondeterminism. For example, consider this monadic version of the Haskell list datatype:

```
data MList m a = MNil | MCons a (m (MList m a))
```

The `MList` type makes sense when `m` is instantiated to lists or trees. But type `MList` has no model when `m` is the set monad: If it did, then we would be able to construct an injective function into a set from its powerset, which is logically unsound.

The set, list, and tree monads all fail to meet the requirements of a powerdomain: The list and tree monads fail to satisfy all of the powerdomain laws, and the operations of the set monad fail to be continuous. In the next section, we will see how to mathematically define type constructors that meet all the requirements of a powerdomain. Defining recursive datatypes with powerdomains is beyond the scope of this chapter, but will be covered later in Chapters 6 and 7.

5.3 POWERDOMAINS

A powerdomain is a type constructor with monadic return and bind operators, and also a binary nondeterministic choice operator. These operations must satisfy all seven powerdomain laws, and furthermore they must all be continuous functions.

Multiple varieties of powerdomains exist that meet all the requirements. 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. Each variety is also traditionally associated with a musical symbol: sharp ($\sharp$) for upper, flat ($\flat$) for lower, and natural ($\natural$) for the convex powerdomain [GS90].

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.

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.

5.3.1 Convex powerdomain

For a given element domain α, the convex powerdomain $\mathcal{P}^\natural(\alpha)$ is specified as the *free continuous domain-algebra*[1] over the constructors $\{-\}^\natural$ and $(\cup^\natural)$, modulo the associativity, commutativity, and idempotence of $(\cup^\natural)$. 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:

$$\frac{\mathrm{adm}(P) \qquad \forall x.\, P(\{x\}^\natural) \qquad \forall xs\; ys.\, P(xs) \longrightarrow P(ys) \longrightarrow P(xs \cup^\natural ys)}{\forall xs.\, P(xs)} \tag{5.8}$$

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. 5.1. Laws 5–7, stating that $(\cup^\natural)$ is associative, commutative, and idempotent, hold

[1] This construction is explained in Abramsky & Jung [AJ94, §6.1].

by construction. We can use laws 1 and 4, which specify how bind interacts with $\{-\}^\natural$ and $(\cup^\natural)$, respectively, as defining equations for the bind operator. Finally, it is straightforward to prove the remaining monad laws (2 and 3) by induction.

Definition 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 5.3.1. *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 because 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 because 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 5.3.1 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 that has exactly two possible return values: either $(\mathit{True}, \mathit{False})$ or $(\bot, \bot)$. Next, define a computation g that additionally has a third possible return value of $(\mathit{True}, \bot)$. Here is how we might specify f and g in Haskell:

```
f, g :: (ChoiceMonad 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 $\{(\mathit{True}, \mathit{False}), (\bot, \bot)\}^\natural$, and the denotation of g is $\{(\mathit{True}, \bot), (\mathit{True}, \mathit{False}), (\bot, \bot)\}^\natural$. But according to Theorem 5.3.1, 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.

By using the convex powerdomain instead of ordinary sets, we pay a price, because the convex powerdomain cannot distinguish as many values. But we get a significant bonus in exchange: Because powerdomains are cpos, and all the operations are continuous, we can freely use powerdomain operations with general recursion—this is not something that can be done with the ordinary set monad.

5.3.2 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:

$$xs \cup^\sharp ys \sqsubseteq xs \tag{5.9}$$

(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 5.3.2. *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. (5.9), 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 because 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 $\perp$ is a member of xs, then everything is a member of xs. In other words, if a nondeterministic computation has any possibility of returning $\perp$, then according to the upper powerdomain semantics, nothing else matters—it might as well *always* return $\perp$. For this reason, the upper powerdomain is good for reasoning about total correctness: if $\perp$ is *not* a member of xs, then you can be sure that xs denotes a computation that has no possibility of nontermination.

5.3.3 Lower powerdomain

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

$$xs \sqsubseteq xs \cup^\flat ys \tag{5.10}$$

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

Theorem 5.3.3. *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 5.3.2. □

An immediate consequence of this theorem is that in the lower powerdomain, $\bot$ is a member of everything. Equivalently, $\{\bot\}^\flat$ 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.

5.3.4 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. 5.2 shows all three powerdomains over a small flat domain, like the lifted booleans. Fig. 5.3 extends this to a slightly larger flat domain. Fig. 5.4 extends this in a different way by adding a top value.

Looking at Figs. 5.2 and 5.3, 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 $\bot$) 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.

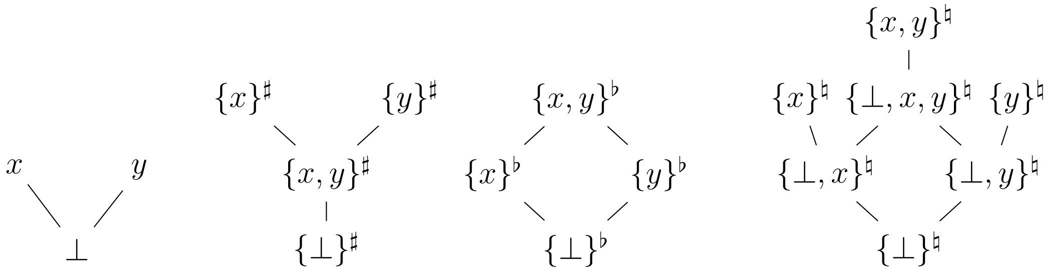


Figure 5.2: Lifted two-element type, with upper, lower, and convex powerdomains

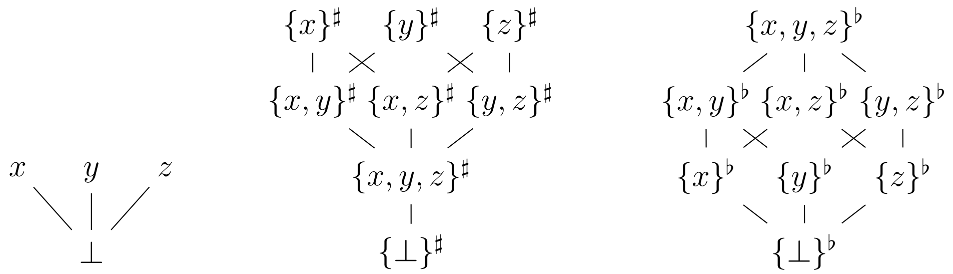


Figure 5.3: Lifted three-element type, with upper and lower powerdomains

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.

5.4 POWERDOMAIN LIBRARY FEATURES

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. 5.5.

The functions convex_to_lower and convex_to_upper are the mappings guaranteed

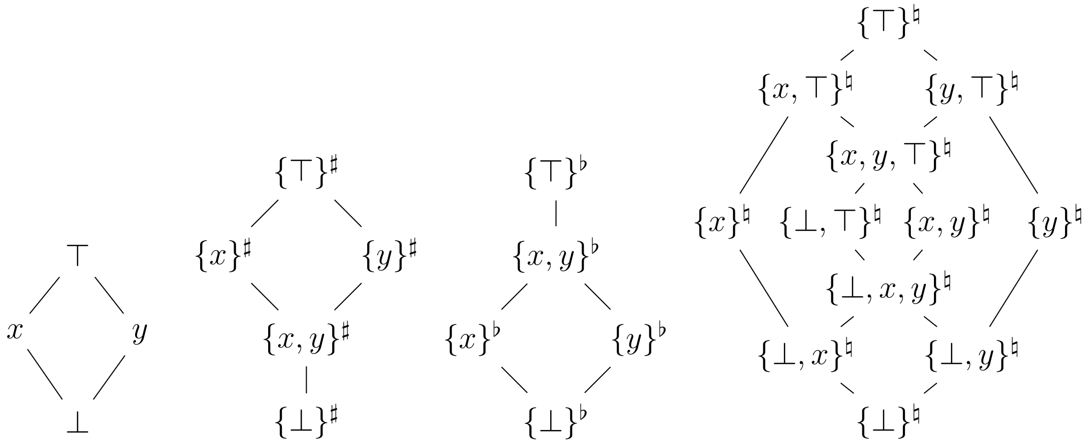


Figure 5.4: Four-element lattice, with upper, lower, and convex powerdomains

to exist by the universal property of the convex powerdomain; they preserve unit and plus. Note that instead of the full function space ($\Rightarrow$), all functions use the HOLCF continuous function space type ($\rightarrow$), indicating that they are continuous functions.

For convenience, the library also provides set-style syntax for powerdomain operations: We can write $\{x\}^\sharp$ for upper_unit·x, xs $\cup^\sharp$ ys for upper_plus·xs·ys, $\bigcup^\sharp$x$\in$xs. t for upper_bind·xs·(Λ x. t), and so on for the other powerdomain types.

Along with the definitions of types and constants, the library provides a significant body of lemmas, many of which are declared to the simplifier. Each powerdomain type has an induction rule in terms of unit and plus, similar to Eq. (5.8). Rules about injectivity, strictness, compactness, and ordering are provided for the constructors. Rewrite rules are provided for the bind, map, and join functions applied to unit, plus, or $\bot$. All of the powerdomain laws are also included as lemmas.

5.4.1 Type class constraints

The main axiomatic type classes in HOLCF are cpo (chain-complete partial orders) and pcpo (pointed cpos). Unfortunately, the powerdomain constructions do not

```
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

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

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
```

Figure 5.5: Powerdomain constants defined in HOLCF '11

work over arbitrary cpos; they need some additional structure. To formalize powerdomains in HOLCF, it was necessary to add a new axiomatic class bifinite, which is a subclass of pcpo. All of the functions defined in the HOLCF '11 powerdomain theories have a bifinite class constraint. The definition and relevant properties of class bifinite will be discussed in Section 5.6.

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. HOLCF '11 provides bifinite class instances for all of its type constructors: 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 also instances of bifinite. The Domain package also generates instances of the bifinite class, when it is used in its definitional mode (see Chapter 6).

5.4.2 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, which are equations like $x + y = y + x$ whose right and left-hand-sides are the same modulo renaming of variables [NPW02]. For any associative-commutative (AC) 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 syntactic term-ordering) [BN98]. Two of the AC rewrites are simply the associativity and commutativity rules. The third is the left-commutativity rule. For normalizing an associative-commutative-idempotent (ACI) operator, we need a total of five rules: the three AC rewrites, plus the idempotency rule, and also (analogous to left-commutativity) left-idempotency.

$$\begin{aligned}
(xs \cup ys) \cup zs &= xs \cup (ys \cup zs) \\
ys \cup xs &= xs \cup ys \\
ys \cup (xs \cup zs) &= xs \cup (ys \cup zs) \\
xs \cup xs &= xs \\
xs \cup (xs \cup ys) &= xs \cup ys
\end{aligned} \tag{5.11}$$

Permutative rewriting using the ACI rules results in a normal form where expressions are nested to the right, and the terms are sorted according to the syntactic term ordering, with no exact duplicates. In HOLCF '11, this normalization can be accomplished for the convex powerdomains by invoking the simplifier with 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 is below another. For each variety of powerdomain, there is a set of rewrite rules that can automatically reduce an inequality on powerdomains down to inequalities on the underlying type.

$$\begin{aligned}
\{x\}^{\sharp} \sqsubseteq \{y\}^{\sharp} &\iff x \sqsubseteq y \\
xs \sqsubseteq (ys \cup^{\sharp} zs) &\iff (xs \sqsubseteq ys) \wedge (xs \sqsubseteq zs) \\
(xs \cup^{\sharp} ys) \sqsubseteq \{z\}^{\sharp} &\iff (xs \sqsubseteq \{z\}^{\sharp}) \vee (ys \sqsubseteq \{z\}^{\sharp})
\end{aligned} \tag{5.12}$$

$$\begin{aligned}
\{x\}^{\flat} \sqsubseteq \{y\}^{\flat} &\iff x \sqsubseteq y \\
(xs \cup^{\flat} ys) \sqsubseteq zs &\iff (xs \sqsubseteq zs) \wedge (ys \sqsubseteq zs) \\
\{x\}^{\flat} \sqsubseteq (ys \cup^{\flat} zs) &\iff (\{x\}^{\flat} \sqsubseteq ys) \vee (\{x\}^{\flat} \sqsubseteq zs)
\end{aligned} \tag{5.13}$$

$$\begin{aligned} \{x\}^\natural \sqsubseteq \{y\}^\natural &\iff x \sqsubseteq y \\ \{x\}^\natural \sqsubseteq (ys \cup^\natural zs) &\iff (\{x\}^\natural \sqsubseteq ys) \wedge (\{x\}^\natural \sqsubseteq zs) \\ (xs \cup^\natural ys) \sqsubseteq \{z\}^\natural &\iff (xs \sqsubseteq \{z\}^\natural) \wedge (ys \sqsubseteq \{z\}^\natural) \end{aligned} \tag{5.14}$$

For the upper and lower powerdomains, each has a set of three rewrite rules that covers all cases of comparisons. For example, simp add: upper_pd_below_simps will rewrite $\{x, y\}^\sharp \sqsubseteq \{y, z\}^\sharp$ into $x \sqsubseteq z \vee y \sqsubseteq z$, using the rules in Eq. (5.12). Similarly, simplification with lower_pd_below_simps uses the rules in Eq. (5.13) to simplify inequalities on lower powerdomains.

For the convex powerdomain, the three rules in Eq. (5.14) 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:

```
lemma convex_pd_below_iff:
  "(xs ⊑ ys) ⟷
     (convex_to_upper·xs ⊑ convex_to_upper·ys ∧
      convex_to_lower·xs ⊑ convex_to_lower·ys)"
```

The rule set convex_pd_below_simps includes all rules from Eqs. (5.12)–(5.14), and a suitably instantiated convex_pd_below_iff 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 simplify with convex_pd_below_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 $y \sqsubseteq z$.

5.5 IDEAL COMPLETION

In Chapter 2, we defined various basic HOLCF types as subsets of other cpos, using the Cpodef package. Unfortunately, this is not possible for powerdomains. In such cases where Cpodef is not applicable, we want to minimize the proof effort for proving the completeness axioms and continuity of operations. One way to accomplish this is to define a cpo using *ideal completion*.

The powerdomain construction used in HOLCF makes use of an alternative representation of cpos, where we just consider the set of compact (i.e., finite) values, rather than the whole cpo [AJ94, §2.2.6]. (Refer to Sec. 2.2.3 for the HOLCF definition and properties of compactness.) For a certain class of cpos, called *algebraic* cpos, every value can be expressed as the least upper bound of its compact approximants. This means that in an algebraic cpo D the set of compact elements $K(D)$, together with the ordering on them, fully represents the entire cpo. We say that $K(D)$ forms a *basis* for the cpo D, and that the entire cpo D is a *completion* of the basis.

To construct a new algebraic cpo by ideal completion, we can start by defining its basis. The ordering on the basis can be any partial order, not necessarily a complete partial order. The operations on the basis only need to be monotone, not necessarily continuous. (This is helpful because monotonicity is generally much easier to prove than continuity.) The ideal completion process extends the basis with new infinite elements to give a cpo. Similarly, a process called *continuous extension* lifts the monotone operations on the basis up to continuous functions on the new cpo.

The ideal completion process is formalized as a library in HOLCF '11; this section will describe the formalization, and show how to define new cpo types with it. Section 5.7 shows how it is used to define the powerdomain type constructors. The process is general enough to be useful for other cpos besides powerdomains;

Chapter 6 will show how it is used to construct a universal domain.

5.5.1 Preorders and ideals

A *preorder* is defined as a binary relation that is reflexive and transitive. Given a basis with a preorder relation $\langle B, \preceq \rangle$, we can construct an algebraic cpo by ideal completion. This is done by considering the set of ideals over the basis:

Definition A set $S \subseteq B$ is an *ideal* with respect to 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 $\{y \mid y \preceq x\}$ for some x, written $\downarrow x$.

The set of all ideals over $\langle B, \preceq \rangle$ is denoted by $\mathrm{Idl}(B)$; 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. The algebraicity of $\mathrm{Idl}(B)$ is manifest in the following induction rule: For an admissible predicate P, if P holds for all principal ideals, then it holds for all elements of $\mathrm{Idl}(B)$.

$$\frac{\mathrm{adm}(P) \qquad \forall x \in B.\, P(\downarrow x)}{\forall y \in \mathrm{Idl}(B).\, P(y)} \tag{5.15}$$

(If the notion of admissibility is defined using directed sets, then Eq. (5.15) holds for any preordered basis B. But if admissibility is defined using countable chains—as it is in HOLCF—then we must require the basis B to be a countable set.)

Note that we do not require $(\preceq)$ 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 quotients by the equivalence induced by $(\preceq)$.

5.5.2 Formalizing ideal completion

Ideal completion is formalized using Isabelle's locale mechanism [KWP99, Bal10]. A *locale* is like a named proof context: It fixes parameters and collects assumptions about them. Lemmas can be proved *in* a locale, where the assumptions of the locale become extra implicit hypotheses. Likewise, constants can be defined in a locale, with the locale parameters as extra implicit arguments. Locales can be *interpreted* by instantiating the parameters with values that satisfy the assumptions, generating specialized versions of all the constants and lemmas from the locale.

Locales are similar in some ways to axiomatic type classes. Both of these mechanisms are used to formalize algebraic structures, which involve some number of fixed operations and assumptions about them. However, each mechanism has its own strengths and limitations, and some situations require one or the other. The formalization of ideal completion relies on two features unique to locales: First, while a type class may only mention a single type variable, locales may be parameterized by any number of types. This feature is necessary because ideal completion relates two types: a basis and a completed cpo. Second, locales allow multiple interpretations at the same type—unlike type classes, which only allow one instantiation per type. This feature allows us to define multiple preorders and ideal completions with the same basis type.

Locale for preorders. The HOLCF '11 ideal completion library defines two locales, preorder and ideal_completion; we will discuss the preorder locale first. The preorder locale fixes a type 'a corresponding to the basis B, and a preorder relation $\preceq$ on that type. We also define a predicate ideal within the locale.

```
locale preorder =
  fixes r :: "'a::type ⇒ 'a ⇒ bool" (infix "≼" 50)
  assumes r_refl: "x ≼ x"
  assumes r_trans: "⟦x ≼ y; y ≼ z⟧ ⟹ x ≼ z"
```

```
definition (in preorder) ideal :: "'a set ⇒ bool"
  where "ideal A ⟷
    (∃x. x ∈ A) ∧ (∀x∈A. ∀y∈A. ∃z∈A. x ≼ z ∧ y ≼ z) ∧
    (∀x y. x ≼ y ⟶ y ∈ A ⟶ x ∈ A)"
```

Within the preorder locale, we prove that principal ideals are indeed ideals. We also prove that the union of a chain of ideals is itself an ideal—which shows that the ideal completion is a cpo.

```
lemma (in preorder) ideal_principal:
  shows "ideal {x. x ≼ z}"

lemma (in preorder) ideal_UN:
  fixes A :: "nat ⇒ 'a set"
  assumes ideal_A: "⋀i. ideal (A i)"
  assumes chain_A: "⋀i j. i ≤ j ⟹ A i ⊆ A j"
  shows "ideal (⋃i. A i)"
```

Next we shall consider the steps required to define a new cpo in Isabelle using ideal completion. The first step is to choose a type to use as a basis and define a preorder relation on it. For example, as a basis we might use lists of natural numbers, with a prefix ordering (@ is Isabelle's list-append operator). After defining the relation we proceed to interpret the preorder locale.

```
definition prefix :: "nat list ⇒ nat list ⇒ bool"
  where "prefix xs ys = (∃zs. ys = xs @ zs)"

interpretation preord_prefix: preorder prefix
  by ...
```

The interpretation command requires a proof that prefix satisfies the assumptions of the locale—in this case, reflexivity and transitivity. After we discharge the proof obligations, the locale package generates copies of all constants and lemmas from the preorder locale, instantiated with prefix in place of $\preceq$ and with the qualifier "preord_prefix" prepended to all the names.

The next step is to define a new type as the set of ideals over the basis, using TYPEDEF. (Recall that the **open** option serves merely to prevent TYPEDEF from defining an unneeded set constant called inflist.) The non-emptiness obligation can be discharged using lemma preord_prefix.ideal_principal.

```
typedef (open) inflist = "{S::nat list set. preord_prefix.ideal S}"
```

After defining the type, we define the ordering (⊑) on type inflist in terms of the subset ordering on type nat list set.

```
instantiation inflist :: below
begin
  definition below_inflist_def: "(x ⊑ y) = (Rep_inflist x ⊆ Rep_inflist y)"
  instance ..
end
```

We still need to prove that inflist is an instance of the po and cpo classes. For this purpose, the ideal completion library provides a pair of lemmas that are very similar to those used by the Cpodef package from Chapter 2 (§2.3). They have assumptions about the type_definition predicate, and their conclusions are OFCLASS predicates.

```
lemma (in preorder) typedef_ideal_po:
  fixes Rep :: "'b::below ⇒ 'a set'' and Abs :: "'a set ⇒ 'b"
  assumes type: "type_definition Rep Abs {S. ideal S}"
  assumes below: "⋀x y. x ⊑ y ⟷ Rep x ⊆ Rep y"
  shows "OFCLASS('b, po_class)"

lemma (in preorder) typedef_ideal_cpo:
  fixes Rep :: "'b::po ⇒ 'a set'' and Abs :: "'a set ⇒ 'b"
  assumes type: "type_definition Rep Abs {S. ideal S}"
  assumes below: "⋀x y. x ⊑ y ⟷ Rep x ⊆ Rep y"
  shows "OFCLASS('b, cpo_class)"
```

The proof of typedef_ideal_po is straightforward. To prove typedef_ideal_cpo, we show that Abs (⋃i. Rep (Y i)) gives the least upper bound for any chain Y.

Using lemma type_definition_inflist (provided by Typedef) and below_inflist_def together with preord_prefix.typedef_ideal_po and preord_prefix.typedef_ideal_cpo, we can prove the po and cpo class instances for inflist.

Locale for ideal completions. Having defined a cpo with ideal completion, we can now define an embedding from the basis type into the completion type, using principal ideals.

```
definition principal_inflist :: "nat list ⇒ inflist"
  where "principal_inflist x = Abs_inflist {a. prefix a x}"
```

In order to prove generic theorems about this embedding, HOLCF defines another locale on top of the preorder locale, called ideal_completion. In addition to type 'a representing the basis B, the new locale fixes another type 'b corresponding to $\mathrm{Idl}(B)$. It fixes two new functions: rep returns the representation of a value as a set of basis elements, generalizing the function Rep_inflist; and principal returns values that correspond to principal ideals, generalizing principal_inflist.

```
locale ideal_completion = preorder +
  fixes principal :: "'a::type ⇒ 'b::cpo"
  fixes rep :: "'b::cpo ⇒ 'a::type set"
  assumes ideal_rep: "⋀x. ideal (rep x)"
  assumes rep_lub: "⋀Y. chain Y ⟹ rep (⨆i. Y i) = (⋃i. rep (Y i))"
  assumes rep_principal: "⋀a. rep (principal a) = {b. b ≼ a}"
  assumes belowI: "⋀x y. rep x ⊆ rep y ⟹ x ⊑ y"
  assumes countable: "∃f::'a ⇒ nat. inj f"
```

The assumptions of the ideal_completion locale are designed to be easily satisfied by types like inflist that are defined by ideal completion over a countable basis type. To assist with ideal_completion locale interpretation proofs, the library provides the following lemma:

```
lemma (in preorder) typedef_ideal_completion:
  fixes Rep :: "'b::cpo ⇒ 'a set" and Abs :: "'a set ⇒ 'b"
  assumes type: "type_definition Rep Abs {S. ideal S}"
  assumes below: "⋀x y. x ⊑ y ⟷ Rep x ⊆ Rep y"
  assumes principal: "⋀a. principal a = Abs {b. b ≼ a}"
  assumes countable: "∃f::'a ⇒ nat. inj f"
  shows "ideal_completion r principal Rep"
```

Within the ideal_completion locale, we start by proving a few simple lemmas about principal: The ordering between principal values reflects the basis ordering, and every principal value is compact.

```
lemma (in ideal_completion) principal_below_iff [simp]:
  "principal a ⊑ principal b ⟷ a ≼ b"
```

```
lemma (in ideal_completion) compact_principal [simp]:
  "compact (principal a)"
```

Perhaps the most important theorem in the ideal_completion locale, however, is the principal induction rule given in Eq. (5.15). In order to help prove it, we must start with a lemma related to the countability of the basis: Any value in the complete cpo can be expressed as the least upper bound of a chain of principal values.

```
lemma (in ideal_completion) obtain_principal_chain:
  "∃Y. (∀i. Y i ≼ Y (Suc i)) ∧ x = (⨆i. principal (Y i))"
```

The proof proceeds by explicitly constructing such a chain, following a technique from the proof of Proposition 2.2.14 in Abramsky and Jung [AJ94]. Let $(b_n)_{n\in\omega}$ be an enumeration of the basis B, and let x be a value in the completion represented by the ideal S. Then we can construct a sequence of basis values $(s_i)_{i\in\omega}$ as follows. Let s_0 be the first b_n such that $b_n \in S$. Then for every $i \in \omega$ we define t_i as the first b_n such that $b_n \in S$ and $b_n \not\preceq s_i$. Then we inductively define s_{i+1} as the first b_n in S above both s_i and t_i. It can be shown that the sequence s_i yields the desired least upper bound.

Using lemma obtain_principal_chain, the principal induction rule follows directly.

```
lemma (in ideal_completion) principal_induct:
  "⟦adm P; ⋀a. P (principal a)⟧ ⟹ P x"
```

As we will see later in Sec. 5.7, induction over principal values is the primary way to transfer properties about the basis type up to the completed cpo. Lemma principal_induct is used dozens of times in the proof scripts of the HOLCF '11 powerdomain theories.

5.5.3 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 $\widehat{f} : \mathrm{Idl}(B) \to C$ that agrees with f on principal ideals, i.e., for all x, $\widehat{f}(\downarrow x) = f(x)$. We say that $\widehat{f}$ is the *continuous extension* of 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: $\widehat{f}(S) = \bigsqcup_{x \in S} f(x)$. Generally, the result type C would need to be a directed-complete partial order[2] to ensure that this least upper bound exists. However, if the basis B is countable, then it is possible to find a chain in S that yields the same least upper bound as S. This means that C can be any chain-complete partial order.

5.5.4 Formalizing continuous extensions

Within the ideal_completion locale we define a function extension, which takes a monotone function f as an argument, and returns the continuous extension $\widehat{f}$.

```
definition (in ideal_completion) extension :: "('a ⇒ 'c::cpo) ⇒ ('b → 'c)"
  where "extension f = (Λ x. lub (image f (rep x)))"
```

The definition of extension uses two features that are only well-defined if certain conditions are met: First, the function lub requires that its argument actually have a least upper bound. Second, the continuous function abstraction requires that the body be continuous in x. Both of these properties rely on the monotonicity of f.

To prove that image f (rep x) has a least upper bound, we use the lemma obtain_principal_chain to get a chain Y :: nat ⇒ 'a of basis elements such that x =

[2] Directed-completeness means that every directed set has a least upper bound. This is a stronger condition than chain-completeness, which is used in the HOLCF formalization of cpos.

(⨆i. principal (Y i)). Then we show that (⨆i. f (Y i)) is the desired least upper bound. The continuity of the abstraction then follows from rep_lub combined with properties of least upper bounds. Finally, we can establish the behavior of extension on principal ideals, using the fact that f a is a least upper bound of the set image f (rep (principal a)).

```
lemma extension_principal:
  assumes f_mono: "⋀a b. a ≼ b ⟹ f a ⊑ f b"
  shows "extension f·(principal a) = f a"
```

To prove a property about a function defined as a continuous extension, the general approach is to use principal induction (lemma principal_induct) to reduce the general subgoal to one about principal values; then lemma extension_principal can be used to unfold the definition.

5.6 BIFINITE CPOS

The construction used here for powerdomains only works with element types that are algebraic cpos, having bases of compact elements. As was mentioned earlier in Sec. 5.4.1, the type classes cpo and pcpo are not sufficient to meet this requirement. Instead, the powerdomain libraries are based on the bifinite class, which is a subclass of pcpo.

Definition A continuous function $f : D \to D$ is a *deflation* if it is idempotent and below the identity function: $f \circ f = f \sqsubseteq \mathrm{Id}_D$. A *finite deflation* is a deflation whose image is a finite set.

Definition Let D be a cpo, and let $\mathcal{M}$ be the set of finite deflations over D. Then we say that D is *bifinite* if $\mathcal{M}$ is countable and directed with $\bigsqcup \mathcal{M} = \mathrm{Id}_D$.

Given a deflation f over a cpo D, the image of f identifies a sub-cpo of D. Similarly, a finite deflation over D identifies a finite poset that is a subset of

$K(D)$. Intuitively then, a bifinite cpo is one that can be expressed as the limit (in an appropriate sense) of a countable collection of finite posets.

A few notes on terminology: The definitions of "deflation" and "finite deflation" used here were taken from Gunter [Gun85, §3.1]. Deflations are also commonly known as "projections" or sometimes "kernel operators" [AJ94]. Abramsky and Jung also use the term "idempotent deflation" to refer to finite deflations [AJ94]. We include a countability requirement in the definition of bifiniteness, following Gunter and Scott [GS90]. Some authors [AJ94] relax this requirement, allowing bifinite domains of arbitrary cardinality, and using the qualifiers "countably based" or "ω-bifinite" as required. Bifinite domains were originally defined by Plotkin as limits of expanding sequences of finite posets, who used the name "SFP domains" [Plo76].

Many categories of cpos can be found in the domain theory literature [AC98, GS90, AJ94]. Of all the possibilities, the bifinites were chosen because they meet the following criteria:

- All bifinite cpos are algebraic: Every bifinite type has a basis of compact elements, given by the union of the ranges of the finite deflations.

- In bifinite cpos, every directed set contains a countable chain with the same limit. This means that for bifinite cpos, 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 class of bifinite cpos is closed under all type constructors used in HOLCF, including all three powerdomains.

- A universal bifinite domain exists, which can be used to represent general recursive datatypes (see Chapter 6).

The requirement for algebraicity rules out the chain-complete cpos (classes cpo and pcpo). The category of countably-based algebraic cpos meets the first two criteria, but it fails the third: The continuous function space between two arbitrary algebraic cpos is not necessarily algebraic. The category of bounded-complete domains (also known as “Scott domains”) meets nearly all of the criteria, including having a universal domain—except that bounded-completeness fails to be preserved by the convex powerdomain [GS90].

5.6.1 Type class for bifinite cpos

Next we will see how the class bifinite is defined in HOLCF ’11. We start by defining deflation and finite_deflation. Defining them as locales makes it more convenient to prove numerous simple lemmas about deflations and finite deflations.

```
locale deflation =
  fixes d :: "'a → 'a"
  assumes idem: "⋀x. d·(d·x) = d·x"
  assumes below: "⋀x. d·x ⊑ x"

locale finite_deflation = deflation +
  assumes finite_fixes: "finite {x. d·x = x}"
```

Note that finite_deflation is defined using the set of fixed points of d, rather than the image of d; it is provable within the deflation locale that these sets are equal. This formulation makes it slightly easier to prove that particular functions are finite deflations.

For class bifinite, instead of asserting directly that the collection of all finite deflations is countable and directed, we assume the existence of a countable chain of them whose least upper bound is the identity. Defining bifiniteness this way in terms of approx_chain is convenient because we will be able to reuse it later: For the universal domain presented in Chapter 6, embedding functions will be constructed within the approx_chain locale.

```
locale approx_chain =
  fixes approx :: "nat ⇒ 'a → 'a"
  assumes chain_approx: "chain (λi. approx i)"
  assumes lub_approx: "(⨆i. approx i) = ID"
  assumes finite_deflation_approx: "⋀i. finite_deflation (approx i)"
```

```
class bifinite = pcpo +
  assumes bifinite: "∃a. approx_chain a"
```

To prove instances of the bifinite class, we rely on a collection of map functions for each type constructor. Map functions were discussed previously in Chapter 4, in the context of the Domain package (Fig. 4.1 gives a complete list). We repeat the definition of the map function for the product type:

```
definition prod_map :: "('a → 'b) → ('c → 'd) → 'a × 'c → 'b × 'd"
  where "prod_map = (Λ f g (x, y). (f·x, g·y))"
```

The prod_map function applied to identity functions yields the identity function on pairs. We can also show that prod_map applied to finite deflations yields a finite deflation.

```
lemma prod_map_ID:
  shows "prod_map·ID·ID = ID"
```

```
lemma finite_deflation_prod_map:
  assumes "finite_deflation d1" and "finite_deflation d2"
  shows "finite_deflation (prod_map·d1·d2)"
```

A consequence of these properties is that prod_map takes approx-chains to approx-chains, which is sufficient to show that the product type constructor preserves bifiniteness. Similarly, other map functions are used to prove bifinite class instances for other basic HOLCF types (strict sums and products, continuous function space, and lifted cpos). Flat lifted HOL types like nat lift are bifinite only if they are countable.

5.6.2 Bifinite types as ideal completions

Every bifinite cpo D has a countable basis $K(D)$ of compact elements, of which D is isomorphic to the ideal completion: $D \cong \mathrm{Idl}(K(D))$. Accordingly, we can create a locale interpretation that lets us treat types in class bifinite as ideal completions. We define a partial order 'a compact_basis isomorphic to the set of compact elements of type 'a. The type 'a compact_basis will serve as the basis $K(D)$ while the original type 'a serves as $\mathrm{Idl}(K(D))$. The Rep_compact_basis function from the type definition fills the role of the function principal from the ideal_completion locale. We define a function approximants to act as the rep function.

```
typedef (open) 'a compact_basis = "{x::('a::bifinite). compact x}"

definition approximants :: "'a::bifinite ⇒ 'a compact_basis set"
  where "approximants x = {a. Rep_compact_basis a ⊑ x}"

interpretation compact_basis:
  ideal_completion below Rep_compact_basis approximants
```

The proof of the locale interpretation is mostly straightforward; the trickiest part is proving that approximants x is a directed set. For the proof, we apply rule bifinite to obtain a chain of approx functions, and use a lemma proved within the approx_chain locale: A value is compact if and only if it is in the range of one of the approx functions. To show directedness of approximants x, let a and b be compact values below x. Then there exist i and j such that approx i·a = a and approx j·b = b. Finally, with a bit more work we can show that approx (max i j)·x is a compact value below x and above a and b.

Countability of the type 'a compact_basis also derives from the compactness rule. The set of compact values of type 'a is the union of the images of approx functions. As a countable union of finite sets, it is also countable.[3]

[3] The proof of this fact requires the axiom of choice. Reliance on AC could be avoided by making an explicit enumeration of the basis part of the bifinite class, but it is not clear that it would be worth the trouble to do so.

5.7 CONSTRUCTION OF POWERDOMAINS

All three of the powerdomains in the library are defined by ideal completion, following the construction given by Gunter and Scott [GS90, §5.2]. Each powerdomain uses the same basis type. If the algebraic cpo D is the element type, then the powerdomain basis consists of nonempty, finite sets of compact elements of D. Formally, we would write this as $\mathcal{P}^*_f(K(D))$, where $K(D)$ is the compact basis of D, and $\mathcal{P}^*_f(S)$ denotes the set of finite, non-empty subsets of set S.

The lower, upper, and convex powerdomains all use the same basis, but each uses a different preorder relation:

$$\begin{aligned}
a \preceq^{\flat} b &\iff \forall x \in a.\ \exists y \in b.\ x \sqsubseteq y \\
a \preceq^{\sharp} b &\iff \forall y \in b.\ \exists x \in a.\ x \sqsubseteq y \qquad (5.16)\\
a \preceq^{\natural} b &\iff a \preceq^{\flat} b \wedge a \preceq^{\sharp} b
\end{aligned}$$

Note that these definitions are consistent with the characteristic ordering properties of powerdomains: We have $a \cup b \preceq^{\sharp} a$ in accordance with Eq. (5.9), and $a \preceq^{\flat} a \cup b$ in accordance with Eq. (5.10).

The various powerdomain operations, including unit, plus, and bind, are all defined as continuous extensions: In particular, the unit operation is the extension of the singleton function $\{-\} : K(D) \to \mathcal{P}^*_f(K(D))$, and plus is the extension of the union operation on $\mathcal{P}^*_f(K(D))$. Properties about all the functions can be derived using principal induction, as in Eq. (5.15).

The remainder of this section shows how these constructions are formalized in HOLCF '11. We start with the powerdomain basis type (§5.7.1), and then define powerdomain types by ideal completion using the various preorders (§5.7.2). Next we define the constructors unit and plus by continuous extension (§5.7.3) and prove properties about them by induction (§5.7.4). Finally we define bind, map, and join functions and prove bifiniteness of the powerdomain types (§5.7.5).

5.7.1 Powerdomain basis type

To use as a basis for the various powerdomains, the library defines a type 'a pd_basis, which consists of nonempty, finite sets of elements of type 'a compact_basis. The constructor functions PDUnit and PDPlus build singleton sets and unions, respectively.

```
typedef 'a pd_basis = "{S::'a compact_basis set. finite S ∧ S ≠ {}}"

definition PDUnit :: "'a compact_basis ⇒ 'a pd_basis"
  where "PDUnit x = Abs_pd_basis {x}"

definition PDPlus :: "'a pd_basis ⇒ 'a pd_basis ⇒ 'a pd_basis"
  where "PDPlus t u = Abs_pd_basis (Rep_pd_basis t ∪ Rep_pd_basis u)"
```

Using the induction principle for finite sets from Isabelle's standard library, we derive an induction rule for the 'a pd_basis type, expressed in terms of the constructors.

```
lemma pd_basis_induct:
  assumes PDUnit: "⋀a. P (PDUnit a)"
  assumes PDPlus: "⋀t u. ⟦P t; P u⟧ ⟹ P (PDPlus t u)"
  shows "P x"
```

For defining operations like map, bind, and join, we also need the following fold operation on type 'a pd_basis. Its definition uses a finite-set fold operator fold1 provided by the standard Isabelle libraries [NP05]. Given an argument built from combinations of PDUnit and PDPlus, the function fold_pd g f replaces PDUnit with g and PDPlus with f—but it is only well-defined if f is associative, commutative, and idempotent.

```
definition fold_pd ::
    "('a compact_basis ⇒ 'b) ⇒ ('b ⇒ 'b ⇒ 'b) ⇒ 'a pd_basis ⇒ 'b"
  where "fold_pd g f t = fold1 f (image g (Rep_pd_basis t))"
```

Finally, using the countability of type 'a compact_basis, we prove that 'a pd_basis is a countable type as well. The proof uses an isomorphism between the natural numbers $\mathbb{N}$ and finite sets of naturals $\mathcal{P}_f(\mathbb{N})$, which is provided in the standard Isabelle libraries.

5.7.2 Defining powerdomain types with ideal completion

We will consider the definition of the upper powerdomain in some detail; the definitions of the other two powerdomain types are very similar. Having already defined the basis type, the next step is to define the preorder relation.

```
definition upper_le :: "'a pd_basis ⇒ 'a pd_basis ⇒ bool" (infix "≤♯" 50)
  where "u ≤♯ v = (∀y ∈ Rep_pd_basis v. ∃x ∈ Rep_pd_basis u. x ⊑ y)"
```

```
interpretation upper_le: preorder upper_le
```

We follow the same process shown above in Sec. 5.5.2 for inflist to define the type, instantiate po and cpo classes, and finally interpret the ideal_completion locale. The complete code for this process is shown in Fig. 5.6.

After proving the class instances and locale interpretations, we get to our first interesting proof: We can prove that type 'a upper_pd is pointed.

```
lemma upper_pd_minimal:
  "upper_principal (PDUnit (Abs_compact_basis ⊥)) ⊑ ys"
```

The proof is by induction on ys using rule upper_pd.principal_induct, which is one of the theorems generated by the locale interpretation.

5.7.3 Defining constructor functions by continuous extension

The extension operators are used to define the powerdomain constructors upper_unit and upper_plus in terms of the singleton and union operations on the pd_basis type. The function upper_unit has an argument type of 'a, which uses 'a compact_basis as its basis type. Accordingly, we must use the extension combinator from the compact_basis locale interpretation to define it.

```
typedef (open) 'a upper_pd = "{S::'a pd_basis set. upper_le.ideal S}"
  by (auto intro: upper_le.ideal_principal)

instantiation upper_pd :: (bifinite) below
begin
  definition "(x ⊑ y) = (Rep_upper_pd x ⊆ Rep_upper_pd y)"
  instance ..
end

instance upper_pd :: (bifinite) po
  using type_definition_upper_pd below_upper_pd_def
  by (rule upper_le.typedef_ideal_po)

instance upper_pd :: (bifinite) cpo
  using type_definition_upper_pd below_upper_pd_def
  by (rule upper_le.typedef_ideal_cpo)

definition upper_principal :: "'a pd_basis ⇒ 'a upper_pd"
  where "upper_principal t = Abs_upper_pd {u. u ≤♯ t}"

interpretation upper_pd: ideal_completion upper_le upper_principal Rep_upper_pd
  using type_definition_upper_pd below_upper_pd_def
  using upper_principal_def pd_basis_countable
  by (rule upper_le.typedef_ideal_completion)
```

Figure 5.6: Defining the upper powerdomain type by ideal completion

```
definition upper_unit :: "'a → 'a upper_pd"
  where "upper_unit =
    compact_basis.extension (λa. upper_principal (PDUnit a))"
```

The next step is to use the theorem compact_basis.extension_principal to establish how upper_unit acts on principal inputs. This requires a proof that the argument to compact_basis.extension is monotone. Monotonicity is easy to show, because a $\sqsubseteq$ b implies PDUnit a $\leq^\sharp$ PDUnit b, which in turn implies upper_principal (PDUnit a) $\sqsubseteq$ upper_principal (PDUnit b).

```
lemma upper_unit_Rep_compact_basis [simp]:
  "upper_unit·(Rep_compact_basis a) = upper_principal (PDUnit a)"
```

Unlike upper_unit, the operator upper_plus takes arguments of type 'a upper_pd, so we must define it using the extension combinator from the upper_pd locale. Because it takes two arguments, we nest two applications of upper_pd.extension.

```
definition upper_plus :: "'a upper_pd → 'a upper_pd → 'a upper_pd"
  where "upper_plus = upper_pd.extension (λt.
    upper_pd.extension (λu. upper_principal (PDPlus t u)))"
```

Again, we prove how the constructor acts on principal inputs by showing monotonicity; here we must prove that the definition is monotone in both arguments. The proof obligation reduces to showing that PDPlus is monotone with respect to $\leq^\sharp$, which is easily proved by unfolding the definitions.

```
lemma upper_plus_principal [simp]:
  "upper_plus·(upper_principal t)·(upper_principal u) =
    upper_principal (PDPlus t u)"
```

As mentioned earlier in Sec. 5.4, we introduce syntax for the constructors: $\{x\}^\sharp$ for upper_unit·x and xs $\cup^\sharp$ ys for upper_plus·xs·ys.

```
lemma upper_plus_assoc: "(xs ∪♯ ys) ∪♯ zs = xs ∪♯ (ys ∪♯ zs)"

lemma upper_plus_commute: "xs ∪♯ ys = ys ∪♯ xs"

lemma upper_plus_absorb: "xs ∪♯ xs = xs"

lemma upper_plus_below1: "xs ∪♯ ys ⊑ xs"

lemma upper_pd_induct:
  assumes P: "adm P"
  assumes unit: "⋀x. P {x}♯"
  assumes plus: "⋀xs ys. ⟦P xs; P ys⟧ ⟹ P (xs ∪♯ ys)"
  shows "P (xs::'a upper_pd)"
```

Figure 5.7: Powerdomain lemmas with simple proofs by principal induction

5.7.4 Proving properties about the constructors

After defining the constructor functions upper_unit and upper_plus, we must prove some properties about them. For example, we need to show that upper_plus satisfies the powerdomain laws of associativity, commutativity, and idempotence. We must also prove the characteristic ordering property of upper powerdomains from Eq. (5.9). These lemmas are all listed in Fig. 5.7, and they all have similar proofs. Because each proposition is admissible in each variable (see Fig. 2.3 from Sec. 2.2.3), we can perform principal induction with rule upper_pd.principal_induct to reduce them to propositions about elements of the basis: After applying induction and simplifying, $(\cup^\sharp)$ is replaced by PDPlus, and $(\sqsubseteq)$ by $(\leq^\sharp)$. The corresponding properties on the basis are then easy to show by unfolding the relevant definitions.

The proof of the induction rule upper_pd_induct starts the same way, with principal induction. This reduces the goal P xs to one of the form P (upper_principal t) for arbitrary t. The proof then proceeds by induction on t using rule pd_basis_induct.

```
lemma upper_unit_below_iff [simp]:
  "{x}♯ ⊑ {y}♯ ⟷ x ⊑ y"

lemma upper_plus_below_unit_iff [simp]:
  "xs ∪♯ ys ⊑ {z}♯ ⟷ xs ⊑ {z}♯ ∨ ys ⊑ {z}♯"
```

Figure 5.8: Powerdomain lemmas with tricky proofs by principal induction

Other properties, such as the rewrite rules for comparisons from Eqs. (5.12)–(5.14), are a bit trickier to prove. Because they contain implications, which do not preserve admissibility, the principal induction rules are not so straightforward to apply. For example, consider the lemma upper_unit_below_iff from Fig. 5.8. One direction of the equivalence can be solved by monotonicity, but this still leaves the implication $\{x\}^\sharp \sqsubseteq \{y\}^\sharp \longrightarrow x \sqsubseteq y$, which is equivalent to $\{x\}^\sharp \not\sqsubseteq \{y\}^\sharp \lor x \sqsubseteq y$. Due to the negated comparison, this predicate is admissible in x but not in y. To complete the proof we will need to perform induction on both variables, but at first it seems we are stuck.

The solution involves using the admissibility rules for compactness from Chapter 2, specifically lemma adm_compact_not_below from Fig. 2.4. Because the proposition is admissible in x, we do principal induction on x first; this replaces each occurrence of x in the goal with Rep_compact_basis a, for an arbitrary a. The new subgoal now looks like this:

$$\{\mathsf{Rep\_compact\_basis}\ a\}^\sharp \sqsubseteq \{y\}^\sharp \longrightarrow \mathsf{Rep\_compact\_basis}\ a \sqsubseteq y$$

The original proposition with x was not admissible in y, but the new proposition is, because $\{\mathsf{Rep\_compact\_basis}\ a\}^\sharp$ is compact (it equals upper_principal (PDUnit a)). This means we can proceed with a second principal induction on y; the remainder of the proof is easy.

We use a similar proof strategy for lemma upper_plus_below_unit_iff. The proposition of that lemma is admissible in xs and ys, but not in z. But after doing principal induction on both xs and ys, replacing them respectively with upper_principal t and upper_principal u, we can use lemma adm_compact_not_below to show that the remaining subgoal is admissible in z. Most of the lemmas corresponding to Eqs. (5.12)–(5.14) use a similar proof, as does the convex_pd_below_iff lemma, which also has an if-and-only-if form.

5.7.5 Defining functor and monad operations

The upper_bind, upper_map, and upper_join operations remain to be defined. Instead of defining each of these separately using continuous extension, it will be easiest to simply define the map and join operations in terms of upper_bind. For the bind operation, we start by defining a function upper_bind_basis that specifies how upper_bind should behave on compact inputs.

```
definition upper_bind_basis ::
   "'a pd_basis ⇒ ('a → 'b upper_pd) → 'b upper_pd"
 where "upper_bind_basis =
   fold_pd (λa. Λ f. f·(Rep_compact_basis a)) (λx y. Λ f. x·f ∪♯ y·f)"
```

We must show that the second argument to fold_pd is associative, commutative, and idempotent before we can derive the characteristic equations. These conditions follow directly from the lemmas shown in Fig. 5.7.

```
lemma upper_bind_basis_simps:
 "upper_bind_basis (PDUnit a) = (Λ f. f·(Rep_compact_basis a))"
 "upper_bind_basis (PDPlus t u) =
   (Λ f. upper_bind_basis t·f ∪♯ upper_bind_basis u·f)"
```

Next, upper_bind is defined as the continuous extension of upper_bind_basis. The proof that upper_bind_basis is monotonic, i.e. that t $\leq^\sharp$ u implies upper_bind_basis t $\sqsubseteq$ upper_bind_basis u, proceeds by induction on u, and relies on upper_plus_below1 from Fig. 5.7.

```
definition upper_bind :: "'a upper_pd → ('a → 'b upper_pd) → 'b upper_pd"
  where "upper_bind = upper_pd.extension upper_bind_basis"
```

After deriving how upper_bind behaves on principal inputs, it is easy to prove how it acts on the upper_unit and upper_plus constructors, using principal induction on the arguments.

```
lemma upper_bind_unit [simp]:
  "upper_bind·{x}♯·f = f·x"
```

```
lemma upper_bind_plus [simp]:
  "upper_bind·(xs ∪♯ ys)·f = upper_bind·xs·f ∪♯ upper_bind·ys·f"
```

Next we can define upper_map in terms of upper_bind and upper_unit. Many properties of upper_map can be derived from related lemmas about upper_bind simply by unfolding the definition.

```
definition upper_map :: "('a → 'b) → 'a upper_pd → 'b upper_pd"
  where "upper_map = (Λ f xs. upper_bind·xs·(Λ x. {f·x}♯))"
```

We will define upper_join similarly in terms of upper_bind, but there is something else we must do first. Note that the argument type of upper_join is a powerdomain of powerdomains. But the type constructor upper_pd is only well-defined when applied to types in the bifinite class. So we must prove the bifiniteness of type 'a upper_pd before we can proceed. To prove that the upper powerdomain is bifinite, we use the same method as with other HOLCF types: We use the map function for the type constructor, and show that it preserves finite deflations (and thus, that it also preserves approx-chains).

```
lemma finite_deflation_upper_map:
  assumes "finite_deflation d" shows "finite_deflation (upper_map·d)"
```

Proving that upper_map preserves deflations is relatively easy; each property of deflations can be proven by induction with rule upper_pd_induct. Proving finiteness of the image is a bit harder. The image of d is a finite set of elements of type 'a, which are all compact. This translates to a finite set of values of type 'a compact_basis. Its powerset then determines a finite subset of 'a pd_basis; in turn, this can be embedded into a finite subset of 'a upper_pd, which can be shown to contain the image of upper_map·d.

After establishing the bifiniteness of the upper powerdomain, we can finally define the join operator.

```
definition upper_join :: "'a upper_pd upper_pd → 'a upper_pd"
  where "upper_join = (Λ xss. upper_bind·xss·(Λ xs. xs))"
```

All of the theorems about map and join, including the monad laws, are proven by induction on their arguments using upper_pd_induct, and simplifying with their definitions.

5.8 DISCUSSION

An earlier version of the work presented in this chapter was published in [Huf08]. The current version includes various simplifications and improvements compared to the earlier work. In the earlier version, the bifinite type class fixed a specific chain of approx functions, rather than just asserting the existence of one. At the time, this was necessary because the proofs of the “tricky” lemmas in Fig. 5.8 used approx functions. The new proofs take full advantage of the latest automation for admissibility proofs involving compactness (see Sec. 2.2.3), and are much simpler.

The ideal_completion locale was also more complicated in the earlier version of the library. Instead of using a countability requirement, it fixed a chain of idempotent take functions over the basis type, corresponding to the chain of approx functions on the completed cpo. With the old version, the ideal completion library was very firmly tied to the definition of the bifinite class. In contrast, the new version of the library is more flexible, and could conceivably be used with any countably-based algebraic cpo. Using ideal completion with unpointed types is planned for future work; this would allow, for example, powerdomains with unpointed element types like the discrete natural numbers.

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 [PM00]. Along similar lines, Thiemann used a type of state monad built on top of powerdomains to reason about concurrent computations [Thi95]. Some of the monad transformers used in these works, specifically the resumption monad transformer, have been studied in previous joint work with Matthews and White [HMW05]. These ideas will be developed further in the case study in Chapter 7, which examines a recursive monadic datatype involving powerdomains.

Another potential application of powerdomains is for modeling exceptions: Peyton Jones, et al. [PJRH$^+$99] use an upper powerdomain to model the meanings of datatypes in a functional language with imprecise run-time exceptions. Even if the execution of a program is actually deterministic in fact, using a less-precise semantics based on powerdomains makes it possible to use a wider range of program optimizations. A transformed program might yield a different run-time exception when executed, yet the transformed program can still be proven to be equivalent in the powerdomain model.

The future work section of [Huf08] noted that the powerdomain library still required integration with the domain package. In HOLCF '11 this integration is complete: In Chapter 6 we will see how the new domain package generates instances of the bifinite class, and how powerdomains are now supported in recursive definitions with the Domain package.

Chapter 6

THE UNIVERSAL DOMAIN AND DEFINITIONAL DOMAIN PACKAGE

6.1 INTRODUCTION

In Chapters 2 and 5 we have seen how to construct a wide variety of type constructors for HOLCF, including strict sums and products, lifted cpos, continuous function spaces, and three kinds of powerdomains. Additionally, the Domain package described in Chapter 4 provides automation for defining *recursive* cpo types, but there are gaps in its implementation: In particular, each new type is not actually *defined*; rather, it is *axiomatized* instead. Generating axioms for each definition leads to serious concerns about soundness.

This chapter describes the formalization of a universal domain that is suitable for modeling general recursive datatypes. The construction is purely definitional, introducing no new axioms. Defining recursive types in terms of this universal domain allows the new Domain package to derive strong reasoning principles, with soundness ensured by construction.

A *universal domain* is a single cpo type that contains a large class of other cpos as subsets. The universal domain presented in this chapter is a universal *bifinite* domain, meaning that any bifinite cpo can be represented within it. (See Chapter 5 for a discussion of bifinite cpos.) More specifically, it is a *deflation*-universal bifinite domain, because each bifinite cpo is represented as a deflation, and type constructors are represented as continuous functions on deflations. The deflation model of recursive datatypes is convenient to work with, because recursive datatypes can be defined with the same least fixed point machinery used for defining recursive functions.

Constructions of a universal bifinite domain exist in the domain theory literature [Gun87, GS90]. This chapter will show how to adapt one such construction so that it can be formalized in a theorem prover like Isabelle. The formalization uses the ideal completion process described previously in Chapter 5 (§5.5).

Contributions. The original contributions presented in this chapter are:

- A new construction of a universal domain that can represent a wide variety of types, including sums, products, continuous function space, powerdomains, and recursive types built from these. Universal domain elements are defined in terms of sets of natural numbers, using ideal completion—thus the construction is suitable for simply-typed, higher-order logic theorem provers.

- A formalization of this construction in the HOLCF library of the Isabelle theorem prover. The formalization is fully definitional; no new axioms are asserted.

- A formalization of a type of algebraic deflations, which are used to represent types and type constructors. As a cpo, this type is also used to build representations of recursive datatypes as least fixed points.

- An extension of the HOLCF Domain package to construct new types explicitly, replacing the isomorphism and reach axioms with actual theorems. The new Domain package is purely definitional, and no longer declares any axioms.

- The definition of a class of *predomain* types, which are an unpointed variant of bifinite domains. We show how predomains can be represented with algebraic deflations, and how support for predomains can be integrated into the new Domain package.

Overview. The remainder of the chapter is organized as follows. We start with background material about embedding-projection pairs and the deflation model of recursive datatypes (§6.2); this material previews the implementation of the definitional Domain package and motivates the definition of a universal domain. Next is a summary of the user-visible interface to the HOLCF universal domain library (§6.3). The construction of the universal domain type itself, along with embedding and projection functions, is covered in the following section (§6.4).

After defining the universal domain, we move on to algebraic deflations, which are used to formalize the class of representable domains (§6.5); and then to the actual implementation of the new definitional Domain package (§6.6). We also describe an unpointed variant of representable domains, called *predomains*, and how they are supported by the Domain package (§6.7). The chapter concludes with a discussion of related work (§6.8).

A significant portion of the material presented in this chapter is based on previously published work. The formalization of the universal domain was initially described in the author's 2009 paper [Huf09a], which predated the completion of the definitional Domain package. Many of the ideas related to representable domains and deflation combinators originated in an earlier joint paper with Matthews and White [HMW05].

6.2 BACKGROUND

6.2.1 Embedding-projection pairs and deflations

Some cpos can be embedded within other cpos. The concept of an *embedding-projection pair* (often shortened to *ep-pair*) formalizes this notion. Let A and B be cpos, and $e : A \to B$ and $p : B \to A$ be continuous functions. Then e and p are an ep-pair if $p \circ e = \mathrm{Id}_A$ and $e \circ p \sqsubseteq \mathrm{Id}_B$. In this case, we write $(e, p) : A \overset{ep}{\to} B$. The existence of such an ep-pair means that cpo A can be embedded in cpo B.

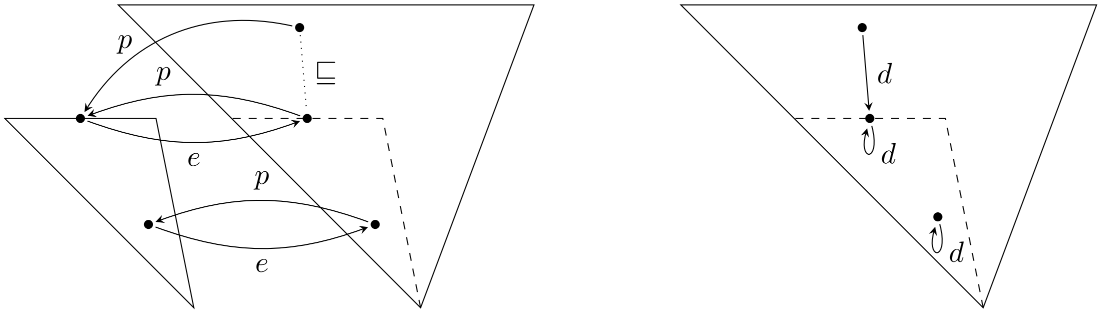
```
lemma ep_pair_comp:
  assumes "ep_pair e1 p1" and "ep_pair e2 p2"
  shows "ep_pair (e2 oo e1) (p1 oo p2)"

definition prod_map :: "('a → 'b) → ('c → 'd) → 'a × 'c → 'b × 'd"
  where "prod_map = (Λ f g p. (f·(fst p), g·(snd p)))"

lemma ep_pair_prod_map:
  assumes "ep_pair e1 p1" and "ep_pair e2 p2"
  shows "ep_pair (prod_map·e1·e2) (prod_map·p1·p2)"

definition cfun_map :: "('b → 'a) → ('c → 'd) → ('a → 'c) → ('b → 'd)"
  where "cfun_map = (Λ a b f x. b·(f·(a·x)))"

lemma ep_pair_cfun_map:
  assumes "ep_pair e1 p1" and "ep_pair e2 p2"
  shows "ep_pair (cfun_map·p1·e2) (cfun_map·e1·p2)"
```

Figure 6.1: Lemmas for composing ep-pairs

```
locale ep_pair =
  fixes e :: "'a → 'b" and p :: "'b → 'a"
  assumes e_inverse: "⋀x. p·(e·x) = x"
  assumes e_p_below: "⋀y. e·(p·y) ⊑ y"
```

Ep-pairs have many useful properties: e is injective, p is surjective, both are strict, each function uniquely determines the other, and the range of e is a sub-cpo of B. The composition of two ep-pairs yields another ep-pair: If $(e_1, p_1) : A \overset{ep}{\to} B$ and $(e_2, p_2) : B \overset{ep}{\to} C$, then $(e_2 \circ e_1, p_1 \circ p_2) : A \overset{ep}{\to} C$. Ep-pairs can also be lifted over many type constructors, including cartesian product and continuous function space (see Fig. 6.1).

A continuous function $d : A \to A$ is a *deflation* if it is idempotent and below the identity function: $d \circ d = d \sqsubseteq \mathrm{Id}_A$.



(a) ep-pair

(b) deflation

Figure 6.2: Embedding-projection pairs and deflations

```
locale deflation =
  fixes d :: "'a → 'a"
  assumes idem: "⋀x. d·(d·x) = d·x"
  assumes below: "⋀x. d·x ⊑ x"
```

Deflations and ep-pairs are closely related. Given an ep-pair $(e, p) : A \overset{ep}{\to} B$, the composition $e \circ p$ is a deflation on B whose image set is isomorphic to A. Conversely, every deflation $d : B \to B$ also gives rise to an ep-pair. Define the cpo A to be the image set of d; also define e to be the inclusion map from A to B, and define $p = d$. Then (e, p) is an embedding-projection pair. So saying that there exists an ep-pair from A to B is equivalent to saying that there exists a deflation on B whose image set is isomorphic to A. Figure 6.2 shows the relationship between ep-pairs and deflations.

A deflation is a function, but it can also be viewed as a set: Just take the image of the function, or equivalently, its set of fixed points—for idempotent functions they are the same. The dashed outline in Fig. 6.2 shows the set defined by the deflation d. Every deflation on a cpo A gives a set that is a sub-cpo, and contains $\bot$ if A has a least element. Not all sub-cpos have a corresponding deflation, but if one exists then it is unique. The set-oriented and function-oriented views of deflations also give the same ordering: For any deflations f and g, $f \sqsubseteq g$ if and only if $\mathrm{Im}(f) \subseteq \mathrm{Im}(g)$.

6.2.2 Deflation model of datatypes

We say that a type A is *representable* in U if there exists an ep-pair from A to U, or equivalently if there exists a deflation d_A on U whose image $\mathrm{Im}(d)$ is isomorphic to A. We say that U is a *universal domain* for some class of cpos if every cpo in the class is representable in U.

While types can be represented by deflations, type *constructors* (which are like functions from types to types) can be represented as functions from deflations to deflations. We say that a type constructor F is representable in U if there exists a continuous function Φ, mapping from deflations to deflations, such that $\mathrm{Im}(\Phi(d))$ is isomorphic to $F(\mathrm{Im}(d))$. Such deflation combinators can be used to build deflations for recursive datatypes [GS90, §7]. The remainder of this section will show how this process works, by example. The new definitional Domain package uses the same process to construct recursive datatypes, as will be explained in Sec. 6.6.

To illustrate the concepts of ep-pairs, deflations, and representable types, we will examine some concrete implementations of these ideas in Haskell. We start by defining a universal Haskell datatype `U`, which should be able to encode every datatype definable in Haskell.

```
data U = Con String [U] | Fun (U -> U)
```

Type `U` has two constructors, `Con` and `Fun`. For encoding ordinary algebraic datatypes like pairs, lists, or trees, we only need to use `Con`. The string identifies the constructor name; the list contains each of its encoded arguments. The constructor `Fun` is required for the function space type `(->)` and other datatypes containing functions.

We then define a Haskell type class `Rep` for "representable" Haskell types. We expect that for each `Rep` class instance, `emb` and `prj` should always form an ep-pair.

```
class Rep a where
  emb :: a -> U
  prj :: U -> a
```

Booleans are representable. Below we make the standard Haskell type `Bool` an instance of class `Rep`. To check that `emb` and `prj` are an ep-pair, we can see that `prj` is the inverse of `emb`. Also, we can see that `prj` is as "undefined" as possible—it maps every ill-formed value to ⊥. Given the definition of `emb`, this is the only definition of `prj` that will yield an ep-pair.

```
instance Rep Bool where
  emb True             = Con "True" []
  emb False            = Con "False" []
  prj (Con "True" [])  = True
  prj (Con "False" []) = False
  prj _                = undefined
```

The Haskell type `Bool` is represented by the function `tBool`, a deflation which is equal to the composition `(emb . prj)` for type `Bool`. For an input that corresponds to a well-formed encoded boolean, `tBool` maps the input to itself. Any other input is mapped to ⊥.

```
tBool :: U -> U
tBool (Con "True" [])  = Con "True" []
tBool (Con "False" []) = Con "False" []
tBool _                = undefined
```

Lists are representable. Next we will consider the standard Haskell list datatype. We will create an ep-pair from type `[a]` to `U` as a composition of two ep-pairs, with `[U]` used as an intermediate type. The functions `embList` and `prjList` form an ep-pair from `[U]` to `U`.

```
embList :: [U] -> U
embList []       = Con "[]" []
embList (x : xs) = Con ":" [x, embList xs]

prjList :: U -> [U]
prjList (Con "[]" [])      = []
prjList (Con ":" [x, xs]) = x : prjList xs
prjList _                 = undefined
```

To build an ep-pair from `[a]` to `[U]`, we can use the `emb` and `prj` functions for the element type `a`, noting that the standard `map` function for lists preserves ep-pairs. Finally, we can define `emb` and `prj` on lists using function composition:

```
instance Rep a => Rep [a] where
  emb = embList . map emb
  prj = map prj . prjList
```

The list type constructor can be represented by the function `tList`, shown below.

```
tList :: (U -> U) -> (U -> U)
tList d = embList . map d . prjList
```

We can see that if `d` is a deflation, then so is `tList d`. It is also easy to verify that `tList` actually does represent the list type constructor: If `d` is equal to `(emb . prj)` for type `a`, then `tList d` is equal to `(emb . prj)` for type `[a]`.

Testing these functions on some example inputs can give us a better idea of how they work. As expected, the deflation `tList tBool` maps any value that corresponds to an encoded list of booleans to itself. For example, `tList tBool` leaves the result of `emb [True, False]` unchanged:

```
emb [True, False] =
  Con ":" [Con "True" [], Con ":" [Con "False" [], Con "[]" []]]

tList tBool
 (Con ":" [Con "True" [], Con ":" [Con "False" [], Con "[]" []]])
 = Con ":" [Con "True" [], Con ":" [Con "False" [], Con "[]" []]]
```

When applied to an ill-formed argument, however, a deflation like `tList tBool` has the effect of replacing any ill-formed portions of the input with `undefined`:

```
tList tBool
  (Con ":" [Con "bogus" [],
            Con ":" [Con "False" [], Con "wrong" []]])
  = Con ":" [undefined, Con ":" [Con "False" [], undefined]]
```

Representability of function space. The approach used for lists can be generalized to any other type constructor with a map function—even contravariant types like the function space. We start by defining functions `embFun` and `prjFun`, which form an ep-pair between types `U -> U` and `U`.

```
embFun :: (U -> U) -> U
embFun !f = Fun f

prjFun :: U -> (U -> U)
prjFun (Fun f) = f
prjFun _ = undefined
```

The strictness annotation (`!f`) is necessary to make `embFun` strict, because Haskell functions with variable patterns are lazy by default. (Both components of an ep-pair must be strict functions.)

Next, we define a `map`-like operator for the Haskell function type `(->)`. The type of the argument `a :: a2 -> a1` reflects the fact that the function space type constructor is contravariant in its left argument.

```
mapFun :: (a2 -> a1) -> (b1 -> b2) -> (a1 -> b1) -> (a2 -> b2)
mapFun a b !f = b . f . a
```

Finally, we can define a class instance for the function type `a -> b`.

```
instance (Rep a, Rep b) => Rep (a -> b) where
  emb = embFun . mapFun prj emb
  prj = mapFun emb prj . prjFun
```

For the list example, `emb` on lists only calls `emb` on the element type, and similarly `prj` calls only `prj`. But because the function type `a -> b` is contravariant in `a`, we have `emb :: (a -> b) -> U` calling `prj :: U -> a`, and `prj :: U -> (a -> b)` calling `emb :: a -> U`.

The function `tFun` represents the Haskell function space type constructor `(->)`. If `a` and `b` are both deflations, then `tFun a b` will be a deflation also.

```
tFun :: (U -> U) -> (U -> U) -> (U -> U)
tFun a b = embFun . mapFun a b . prjFun
```

Recursive definitions of deflations. Using the deflation combinators tBool, tList, and tFun, we can recursively define new deflations that represent recursive datatypes. For example, consider the deflations tD and tE below:

```
tD, tE :: U -> U
tD = tFun tBool (tList tD)
tE = tFun (tList tE) tE
```

The image set of tD is a cpo D that satisfies the equation D = Bool -> [D]. Likewise, the image set of tE is a cpo E that satisfies E = [E] -> E. In general, we can recursively define a deflation for any given type equation of the form $T = F(T)$, where $F(T)$ is a type expression involving type constructors (like Bool, [], or (->)) for which we have deflation combinators.

Thus by using the deflation model, we can solve type equations with ordinary recursive value definitions. This is a significant benefit for HOLCF, because it means that the theory of least fixed points used by the FIXREC package can all be reused in the new definitional DOMAIN package. As a prerequisite for solving type equations, the DOMAIN package needs deflation combinators for all the basic HOLCF types. And as we have seen above, defining the deflation combinators will require ep-pairs from each basic HOLCF type into the universal domain.

6.3 UNIVERSAL DOMAIN LIBRARY FEATURES

The universal domain library is large (about 1000 lines of definitions and proof scripts), defining various types and numerous functions and constants. However, most of these are for internal use only. Just a few parts of the library are directly relevant for users: There is the universal domain type udom, which is an instance of the bifinite class, along with the following three functions:

```
udom_emb :: "(nat ⇒ 'a → 'a) ⇒ 'a → udom"
udom_prj :: "(nat ⇒ 'a → 'a) ⇒ udom → 'a"
udom_approx :: "nat ⇒ udom → udom"
```

The functions udom_emb and udom_prj give an ep-pair from type 'a to udom. They are parameterized by a chain of approx functions on type 'a.

```
lemma ep_pair_udom:
  assumes "approx_chain a"
  shows "ep_pair (udom_emb a) (udom_prj a)"
```

Recall the definition of approx-chains, used in the previous chapter for defining the class of bifinite cpos. A cpo is bifinite if an approx-chain exists for that type.

```
locale approx_chain =
  fixes approx :: "nat ⇒ 'a → 'a"
  assumes chain_approx: "chain (λi. approx i)"
  assumes lub_approx: "(⨆i. approx i) = ID"
  assumes finite_deflation_approx: "⋀i. finite_deflation (approx i)"
```

In order to help build approx-chains for other types, the universal domain library provides udom_approx, which is an approx-chain for type udom.

```
lemma udom_approx: "approx_chain udom_approx"
```

We can use all three udom functions in combination to yield ep-pairs for types like udom × udom. Using udom_approx with the map combinator prod_map, which preserves finite deflations, yields an approx-chain on type udom × udom. Then the udom_emb and udom_prj functions give the desired ep-pair.

```
definition prod_emb :: "udom × udom → udom"
  where "prod_emb =
    udom_emb (λi. prod_map·(udom_approx i)·(udom_approx i))"

definition prod_prj :: "udom → udom × udom"
  where "prod_prj = udom_prj (λi. prod_map·(udom_approx i)·(udom_approx i))"

lemma ep_pair_prod: "ep_pair prod_emb prod_prj"
```

Embedding-projection pairs are defined similarly for all of the other type constructors defined in HOLCF: $\rightarrow$, $\otimes$, $\oplus$, $-_\perp$, $(-)^\sharp$, $(-)^\flat$, and $(-)^\natural$. These ep-pairs are used to define deflation combinators that represent each type constructor; in turn, these combinators are used to construct solutions to recursive domain equations. Details of this process will be covered fully in Sections 6.5 and 6.6.

6.4 CONSTRUCTION OF THE UNIVERSAL DOMAIN

Informally, a *bifinite domain* is a cpo that can be written as the limit of a sequence of finite partial orders. This section describes how to construct a *universal* bifinite domain $\mathcal{U}$, along with an ep-pair from another arbitrary bifinite domain D into $\mathcal{U}$. The general strategy is as follows:

- From the bifinite structure of D, obtain a sequence of finite posets P_n whose limit is D.
- Following Gunter [Gun87], decompose the sequence P_n further into a sequence of *increments* that insert new elements one at a time.
- Construct a universal domain basis that can encode any increment.
- Construct the actual universal domain $\mathcal{U}$ using ideal completion.
- Define the embedding and projection functions between D and $\mathcal{U}$ using continuous extension, in terms of their action on basis elements.

The process of constructing a sequence of increments is described in Sec. 6.4.1. The underlying theory is standard, so the section is primarily exposition; the original contribution here is the formalization of that work in a theorem prover. The remainder of the construction, including the basis and embedding/projection functions, is covered in Sec. 6.4.2 onwards; here both the theory and the formalization are original.

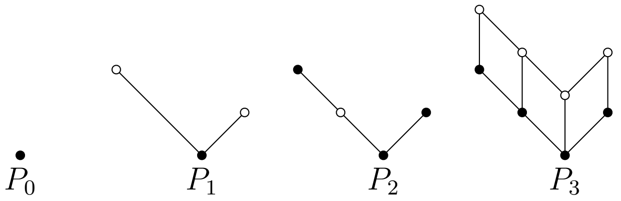


Figure 6.3: A sequence of finite posets. Each P_n can be embedded into P_{n+1}; black nodes indicate the range of the embedding function.

6.4.1 Building a sequence of increments

Any bifinite domain D can be represented as the limit of a sequence of finite posets, with embedding-projection pairs between each successive pair. Figure 6.3 shows the first few posets from one such sequence.

In each step along the chain, each new poset P_{n+1} is larger than the previous P_n by some finite amount; the structure of P_{n+1} has P_n embedded within it, but it has some new elements as well.

An ep-pair between finite posets P and P', where P' has exactly one more element than P, is called an *increment* (terminology due to Gunter [Gun92]). In Fig. 6.3, the embedding of P_1 into P_2 is an example of an increment.

The strategy for embedding a bifinite domain into the universal domain is built around increments. The universal domain is designed so that if a finite partial order P is representable (i.e., by a deflation), and there is an increment from P to P', then P' will also be representable.

For all embeddings from P_n to P_{n+1} that add more than one new value, we will need to decompose the single large embedding into a sequence of smaller increments. The challenge, then, is to determine in which order the new elements should be inserted. The order matters: Adding elements in the wrong order can cause problems, as shown in Fig. 6.4.

To describe the position of a newly-inserted element, it will be helpful to invent

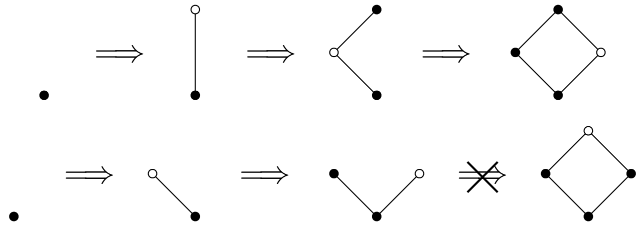


Figure 6.4: The right (top) and wrong (bottom) way to order insertions. No ep-pair exists between the 3-element and 4-element posets on the bottom row.

some terminology. The set of elements *above* the new element will be known as its *superiors*. An element immediately *below* the new element will be known as its *subordinate*. (These terms are not in standard usage.)

In order for the insertion of a new element to be a valid increment, it must have exactly one subordinate. The subordinate indicates the value that the increment's projection maps the new value onto.

With the four-element poset in Fig. 6.4, it is not possible to insert the top element last. The reason is that the element has two subordinates: If a projection function maps the new element to one, the ordering relation with the other will not be preserved. Thus a monotone projection does not exist.

A strategy for successfully avoiding such situations is to always insert maximal elements first [Gun87, §5]. Fig. 6.5 shows this strategy in action. Notice that the number of superiors varies from step to step, but each inserted element always has exactly one subordinate. To maintain this invariant, the least of the four new values must be inserted last.

Armed with this strategy, we can finally formalize the complete sequence of increments for type D. To each element x of the basis of D we must assign a sequence number $place(x)$—this numbering tells in which order to insert the values. The HOLCF formalization breaks up the definition of *place* as follows. First, each basis value is assigned to a rank, where $rank(x) = n$ means that the

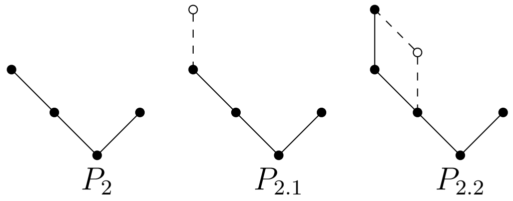


Figure 6.5: A sequence of four increments going from P_2 to P_3. Each new node may have any number of upward edges, but only one downward edge.

basis value x first appears in the poset P_n. Equivalently, $\mathit{rank}(x)$ is the least n such that $\mathit{approx}_n(x) = x$, where approx_n is the finite deflation on D with image P_n. Then an auxiliary function *pos* assigns sequence numbers to values in finite sets, by repeatedly removing an arbitrary maximal element until the set is empty. Finally, $\mathit{place}(x)$ is defined as the sequence number of x within its (finite) rank set, plus the total cardinality of all earlier ranks.

$$\mathit{choose}(A) = (\varepsilon x \in A.\ \forall y \in A.\ x \sqsubseteq y \longrightarrow x = y) \tag{6.1}$$

$$\mathit{pos}(A, x) = \begin{cases} 0, & \text{if } x = \mathit{choose}(A) \\ 1 + \mathit{pos}(A - \{\mathit{choose}(A)\}, x), & \text{if } x \neq \mathit{choose}(A) \end{cases} \tag{6.2}$$

$$\mathit{place}(x) = \mathit{pos}(\{y \mid \mathit{rank}(y) = \mathit{rank}(x)\}, x) + \|\{y \mid \mathit{rank}(x) < \mathit{rank}(y)\}\| \tag{6.3}$$

For the remainder of this chapter, it will be sufficient to note that the *place* function satisfies the following two properties:

Theorem 6.4.1. *Values in earlier ranks come before values in later ranks: If $\mathit{rank}(x) < \mathit{rank}(y)$, then $\mathit{place}(x) < \mathit{place}(y)$.*

Theorem 6.4.2. *Within the same rank, larger values come first: If $\mathit{rank}(x) = \mathit{rank}(y)$ and $x \sqsubseteq y$, then $\mathit{place}(y) < \mathit{place}(x)$.*

6.4.2 A basis for the universal domain

Constructing a partial order incrementally, there are two possibilities for any newly inserted value:

- The value is the very first one (i.e., it is $\bot$)
- The value is inserted above some previous value (its subordinate), and below zero or more other previous values (its superiors)

Accordingly, we can define a recursive datatype B to describe the position of these values relative to each other.

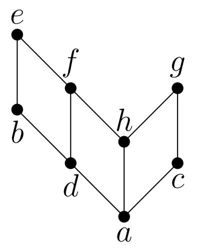
$$B := \bot \mid \langle i, a, S \rangle, \text{where } i \in \mathbb{N},\ a \in B, \text{ and } S \in \mathcal{P}_f(B) \tag{6.4}$$

The notation $\langle i, a, S \rangle$ indicates the value with serial number i, subordinate a, and the finite set of superiors S. (The serial number allows us to distinguish between subsequent values inserted in similar positions.)

The above definition of B does not work as a datatype definition in Isabelle/HOL, because the finite set type constructor does not work with the datatype package. (Indirect recursion only works with other inductive datatypes.) But it turns out that we do not need the datatype package at all—the type B is actually isomorphic to the natural numbers. Using the bijections $\mathbb{N} \cong 1+\mathbb{N}$ and $\mathbb{N} \cong \mathbb{N}\times\mathbb{N}$ with $\mathbb{N} \cong \mathcal{P}_f(\mathbb{N})$, we can construct a bijection that lets us use $\mathbb{N}$ as the basis datatype:

$$\mathbb{N} \cong 1 + \mathbb{N} \times \mathbb{N} \times \mathcal{P}_f(\mathbb{N}) \tag{6.5}$$

Figure 6.6 shows how this system works for embedding all the elements from the poset P_3 into the basis datatype. The elements have letter names from a–h, assigned alphabetically by insertion order. In the datatype encoding of each element, the subordinate and superiors are selected from the set of previously inserted elements. Serial numbers are assigned sequentially.



$$\begin{array}{rcl}
a & = & \bot \\
b & = & \langle 1, a, \{\} \rangle \\
c & = & \langle 2, a, \{\} \rangle \\
d & = & \langle 3, a, \{b\} \rangle \\
e & = & \langle 4, b, \{\} \rangle \\
f & = & \langle 5, d, \{e\} \rangle \\
g & = & \langle 6, c, \{\} \rangle \\
h & = & \langle 7, a, \{e, f, g\} \rangle
\end{array}$$

Figure 6.6: Embedding elements of P_3 into the universal domain basis.

The serial number is necessary to distinguish multiple values that are inserted in the same position. For example, in Fig. 6.6, elements b and c both have a as the subordinate, and neither has any superiors. The serial number is the only way to tell such values apart.

Note that the basis datatype seems to contain some junk—some subordinate/ superiors combinations are not well formed. For example, in any valid increment, all of the superiors are positioned above the subordinate. One way to take care of this requirement would be to define a well-formedness predicate for basis elements. However, it turns out that it is possible (and indeed easier) to simply ignore any invalid elements. In the set of superiors, only those values that are above the subordinate will be considered. (This will be important to keep in mind when we define the basis ordering relation.)

There is also a possibility of multiple representations for the same value. For example, in Fig. 6.6 the encoding of h is given as $\langle 7, a, \{e, f, g\} \rangle$, but the representation $\langle 7, a, \{f, g\} \rangle$ would work just as well (because the sets have the same upward closure). One could consider having a well-formedness requirement for the set of superiors to be upward-closed. But this turns out not to be necessary, because the extra values do not cause problems for any of the formal proofs.

6.4.3 Basis ordering relation

To perform the ideal completion, we need to define a preorder relation on the basis. The basis value $\langle i, a, S\rangle$ should fall above a and below all the values in set S that are above a. Accordingly, we define the relation $(\preceq)$ as the smallest reflexive, transitive relation that satisfies the following two introduction rules:

$$a \preceq \langle i, a, S\rangle \tag{6.6}$$

$$a \preceq b \wedge b \in S \Longrightarrow \langle i, a, S\rangle \preceq b \tag{6.7}$$

Note that the relation $(\preceq)$ is not antisymmetric. For example, we have both $a \preceq \langle i, a, \{a\}\rangle$ and $\langle i, a, \{a\}\rangle \preceq a$. However, for ideal completion this does not matter. Basis values a and $\langle i, a, \{a\}\rangle$ generate the same principal ideal, so they will be identified as elements of the universal domain.

Also note the extra hypothesis $a \preceq b$ in Eq. (6.7). Because we have not banished ill-formed subordinate/superiors combinations from the basis datatype, we must explicitly consider only those elements of the set of superiors that are above the subordinate.

6.4.4 Building the embedding and projection

In the HOLCF formalization, the embedding function *emb* from D to $\mathcal{U}$ is defined using continuous extension. We start by defining *emb* on basis elements, generalizing the pattern shown in Fig. 6.6. The definition below uses wellfounded recursion—all recursive calls to *emb* are on values with smaller *place* numbers.

$$\begin{aligned}
emb(x) &= \begin{cases} \bot & \text{if } x = \bot \\ \langle i, a, S\rangle & \text{otherwise} \end{cases} \\
\text{where } i &= place(x) \\
a &= emb(sub(x)) \\
S &= \{emb(y) \mid place(y) < place(x) \wedge x \sqsubseteq y\}
\end{aligned} \tag{6.8}$$

The subordinate value a is computed using a helper function sub, which is defined as $\mathit{sub}(x) = \mathit{approx}_{n-1}(x)$, where $n = \mathit{rank}(x)$. The ordering produced by the place function ensures that no previously inserted value with the same rank as x will be below x. Therefore the previously inserted value immediately below x must be $\mathit{sub}(x)$, which comes from the previous rank.

In order to complete the continuous extension, it is necessary to prove that the basis embedding function is monotone. That is, we must show that for any x and y in the basis of D, $x \sqsubseteq y$ implies $\mathit{emb}(x) \preceq \mathit{emb}(y)$. The proof is by well-founded induction over the maximum of $\mathit{place}(x)$ and $\mathit{place}(y)$. There are two main cases to consider:

- Case $\mathit{place}(x) < \mathit{place}(y)$: Because $x \sqsubseteq y$, it must be the case that $\mathit{rank}(x) < \mathit{rank}(y)$. Then, using the definition of sub it can be shown that $x \sqsubseteq \mathit{sub}(y)$; thus by the inductive hypothesis we have $\mathit{emb}(x) \preceq \mathit{emb}(\mathit{sub}(y))$. Also, from Eq. (6.6) we have $\mathit{emb}(\mathit{sub}(y)) \preceq \mathit{emb}(y)$. Finally, by transitivity we have $\mathit{emb}(x) \preceq \mathit{emb}(y)$.

- Case $\mathit{place}(y) < \mathit{place}(x)$: From the definition of sub we have $\mathit{sub}(x) \sqsubseteq x$. By transitivity with $x \sqsubseteq y$ this implies $\mathit{sub}(x) \sqsubseteq y$; therefore by the inductive hypothesis we have $\mathit{emb}(\mathit{sub}(x)) \preceq \mathit{emb}(y)$. Also, using Eq. (6.8), we have that $\mathit{emb}(y)$ is one of the superiors of $\mathit{emb}(x)$. Ultimately, from Eq. (6.7) we have $\mathit{emb}(x) \preceq \mathit{emb}(y)$.

The projection function prj from $\mathcal{U}$ to D is also defined using continuous extension. The action of prj on basis elements is specified by the following recursive definition:

$$\mathit{prj}(a) = \begin{cases} \mathit{emb}^{-1}(a) & \text{if } \exists x.\ \mathit{emb}(x) = a \\ \mathit{prj}(\mathit{subordinate}(a)) & \text{otherwise} \end{cases} \tag{6.9}$$

To ensure that prj is well-defined, there are a couple of things to check. First of all, the recursion always terminates: In the worst case, repeatedly taking the subordinate of any starting value will eventually yield $\bot$, at which point the first branch will be taken because $emb(\bot) = \bot$. Secondly, note that emb^{-1} is uniquely defined, because *emb* is injective. Injectivity of *emb* is easy to prove, because each embedded value has a different serial number.

Just like with *emb*, we also need to prove that the basis projection function *prj* is monotone. That is, we must show that for any a and b in the basis of $\mathcal{U}$, $a \preceq b$ implies $prj(a) \sqsubseteq prj(b)$. Remember that the basis preorder $(\preceq)$ is an inductively defined relation; accordingly, the proof proceeds by induction on $a \preceq b$. Compared to the proof of monotonicity for *emb*, the proof for *prj* is relatively straightforward; details are omitted here.

Finally, we must prove that *emb* and *prj* form an ep-pair. The proof of $prj \circ emb = \mathrm{Id}_D$ is easy: Let x be any value in the basis of D. Then using Eq. (6.9), we have $prj(emb(x)) = emb^{-1}(emb(x)) = x$. Because this equation is an admissible predicate on x, proving it for compact x is sufficient to show that it holds for all values in the ideal completion.

The proof of $emb \circ prj \sqsubseteq \mathrm{Id}_U$ takes a bit more work. As a lemma, we can show that for any a in the basis of $\mathcal{U}$, $prj(a)$ is always equal to $emb^{-1}(b)$ for some $b \preceq a$ that is in the range of *emb*. Using this lemma, we then have $emb(prj(a)) = emb(emb^{-1}(b)) = b \preceq a$. Finally, using admissibility, this is sufficient to show that $emb(prj(a)) \sqsubseteq a$ for all a in $\mathcal{U}$.

6.4.5 Bifiniteness of the universal domain

To show that the universal domain $\mathcal{U}$ is bifinite, we must construct a chain of finite deflations on $\mathcal{U}$ whose least upper bound is the identity function. Like all other functions in the universal domain library, these are defined by continuous extension. The action of $uapprox_n$ on basis elements is defined using exactly the same form of recursion as *prj*: We repeatedly take the subordinate of the input value, until the value satisfies some stopping criterion. With *prj*, the criterion was membership in the image of *emb*; for $uapprox_n$, the criterion is that the input, considered as a natural number, is less than or equal to n.

$$uapprox_n(a) = \begin{cases} a & \text{if } a \leq n \\ uapprox_n(subordinate(a)) & \text{otherwise} \end{cases} \tag{6.10}$$

It is straightforward to show that $uapprox_n$ is a deflation, using induction over basis elements. Furthermore, we can show that the image of each $uapprox_n$ equals the set of basis elements corresponding to natural numbers $\{0..n\}$, which is a finite set. To see that $\bigsqcup_n uapprox_n$ is the identity function, consider that for any basis value a, there exists n such that $uapprox_n(a) = a$ (namely, $n = a$).

6.4.6 Implementation in HOLCF

The universal domain type $\mathcal{U}$ is formalized in HOLCF '11 as the type udom. It is defined by ideal completion over the natural numbers, using the method described in Chapter 5. The complete formal proof scripts can be found in the theory file `HOLCF/Universal.thy` of the Isabelle 2011 distribution.

Most of the theory leading up to the *emb* and *prj* functions is parameterized by the chain $approx_n$ of finite deflations, using a copy of the approx_chain locale (see Sec. 5.6).

```
locale approx_chain =
  fixes approx :: "nat ⇒ 'a → 'a"
  assumes chain_approx: "chain (λi. approx i)"
  assumes lub_approx: "(⨆i. approx i) = ID"
  assumes finite_deflation_approx: "⋀i. finite_deflation (approx i)"
```

The HOLCF versions of the functions *choose*, *pos*, *rank*, *place*, *sub*, *emb* and *prj* are all defined within this locale. To generate type-specific *emb* and *prj* functions, we could then perform locale interpretations (see Sec. 5.5.2) for specific approx-chains. However, locale interpretations would be rather wasteful: Each one would unnecessarily create copies of every constant and lemma used in the entire construction, when we really only need two constants (*emb* and *prj*) and one lemma (stating that they form an ep-pair). As a more lightweight alternative, we define udom_emb and udom_prj as the parameterized versions of *emb* and *prj* exported from the locale.

6.5 ALGEBRAIC DEFLATIONS

For solving the domain equations that arise in recursive datatype definitions, we need a cpo $\mathcal{T}$ whose values will represent bifinite domains. For this purpose, we use the set of *algebraic deflations* over the universal domain $\mathcal{U}$: We say that a deflation is algebraic if its image set is an algebraic cpo. In HOLCF '11, the algebraic deflations are defined using ideal completion from the set of *finite deflations*, which have finite image sets. Note that as an ideal completion, $\mathcal{T}$ is itself a cpo; this is important because it lets us use a fixed point combinator to define recursive values of type $\mathcal{T}$, representing recursive types.

6.5.1 Limitations of ordinary deflations

In an earlier HOLCF formalization of representable types and type constructors [HMW05], datatypes were represented using a type of all deflations over a given cpo:

```
pcpodef 'a deflation = "{f::'a → 'a. deflation f}"
```

This definition yields a pointed cpo, because the deflation predicate is admissible, and holds for $\bot$. As a pointed cpo, the deflation type supports a fixed point combinator, which allows us to define recursive deflations to represent recursive datatypes. Furthermore, being defined with the CPODEF package makes it easy to define deflation combinators: We can use the Rep_deflation and Abs_deflation functions, which CPODEF has proved to be continuous.

```
definition prod_deflation ::
    "udom deflation → udom deflation → udom deflation"
  where "prod_deflation = (Λ a b. Abs_deflation
    (prod_emb oo prod_map·(Rep_deflation a)·(Rep_deflation b) oo prod_prj))"
```

In the earlier formalization [HMW05], we defined a class rep of representable domains: These are pointed cpos that can be embedded, via an embedding-projection pair, into the universal domain.

```
class rep = pcpo +
  fixes emb :: "'a → udom"
  fixes prj :: "udom → 'a''
  assumes ep_pair_emb_prj: "ep_pair emb prj"
```

We can obtain the representation of any type in class rep as a value of type udom deflation, by composing emb and prj. (The argument type 'a itself is predefined in Isabelle for use in definitions like this, where the right-hand side mentions a type variable 'a, but no actual values of type 'a. It has a single value, written TYPE('a).)

```
definition rep :: "('a::rep) itself ⇒ udom deflation"
  where "rep (_ :: 'a itself) =
    Abs_deflation ((emb :: 'a → udom) oo (prj :: udom → 'a))"
```

The main problem with this approach is that class rep is not a subclass of class bifinite: Although udom is algebraic, this does not imply that every type representable in udom is algebraic. This means that if we used the above definition for a class of representable domains, then we could not use such types with the powerdomain library, which requires algebraic element types. To ensure that every representable domain is bifinite, we must restrict our attention to the algebraic deflations.

6.5.2 Type of algebraic deflations

As noted above, the HOLCF type of algebraic deflations is defined as an ideal completion from the set of finite deflations. The ideal completion process described in Chapter 5 requires a type to use as a basis; we define the type 'a fin_defl of finite deflations over 'a for this purpose (using the **open** option, as usual, to avoid defining an extra set constant). We then proceed to define the cpo 'a defl of algebraic deflations over 'a, using the same standard process used previously for powerdomains: First define a new type as the set of ideals; then after proving that it is a cpo, define a principal function and interpret the ideal_completion locale.

```
typedef (open) 'a fin_defl = "{d::'a → 'a. finite_deflation d}"

typedef (open) 'a defl = "{S :: 'a fin_defl set. below.ideal S}"

definition defl_principal :: "'a fin_defl ⇒ 'a defl"
  where "defl_principal t = Abs_defl {u. u ⊑ t}"

interpretation defl: ideal_completion below defl_principal Rep_defl
```

The most common operation on deflations is to apply them as functions. For this purpose we define the cast operator. For a deflation t that represents a type, cast·t is essentially like a type cast into that type. As type 'a defl is an ideal completion, we can define cast as the continuous extension of Rep_fin_defl.

```
definition cast :: "'a defl → 'a → 'a"
  where "cast = defl.extension (Rep_fin_defl :: 'a fin_defl ⇒ 'a → 'a)"
```

We can prove a few properties about cast using principal induction. First, that cast always yields deflations; second, that cast preserves ordering.

```
lemma deflation_cast: "deflation (cast·t)"

lemma cast_below_cast: "cast·t ⊑ cast·u ⟷ t ⊑ u"
```

The proof of cast_below_cast is similar to many of the proofs of if-and-only-if lemmas from Chapter 5: The proposition as stated is admissible in t, but not in u. However, after performing principal induction on t, reasoning about compactness shows that the remaining subgoal is admissible in u. The proof also relies on the fact that finite deflations are compact elements of the continuous function space. A direct corollary is that the cast function is injective, which gives us a way to prove that two given algebraic deflations are equal.

6.5.3 Combinators for algebraic deflations

Recall the way that Haskell deflation combinators were defined above in Sec. 6.2.2. Each deflation combinator is written as a composition involving an ep-pair and a map function. For example the combinator tFun, which represents the function space type, is defined like this:

```
tFun :: (U -> U) -> (U -> U) -> (U -> U)
tFun a b = embFun . mapFun a b . prjFun
```

We would like to formalize the Haskell function tFun in HOLCF as a function cfun_defl :: udom defl → udom defl → udom defl. We already know how to formalize the other Haskell functions used in the definition of tFun: First, the map combinator cfun_map can represent mapFun. Then as described in Sec. 6.3, we can use the universal domain library to define cfun_emb and cfun_prj as an ep-pair from udom → udom into udom, to model embFun and prjFun. Our remaining task is then to define cfun_defl so that it meets this specification:

```
lemma cast_cfun_defl:
  "cast·(cfun_defl·a·b) = cfun_emb oo cfun_map·(cast·a)·(cast·b) oo cfun_prj"
```

As cfun_defl is a function that takes algebraic deflations as arguments, we can define it as a continuous extension. Because we will have similar definitions for several other HOLCF type constructors besides the continuous function space, we define a generic combinator defl_fun2 that takes the emb, prj, and map functions as parameters.

```
definition defl_fun2 ::
  "('c → 'u) ⇒ ('u → 'c) ⇒ (('a → 'a) → ('b → 'b) → ('c → 'c))
    ⇒ 'a defl → 'b defl → 'u defl"
 where "defl_fun2 e p f =
  defl.extension (λa. defl.extension (λb. defl_principal
   (Abs_fin_defl (e oo f·(Rep_fin_defl a)·(Rep_fin_defl b) oo p))))"
```

```
definition cfun_defl :: "udom defl → udom defl → udom defl"
 where "cfun_defl = defl_fun2 cfun_emb cfun_prj cfun_map"
```

We can then prove the lemma cast_cfun_defl as an instance of a generic lemma about defl_fun2. The proof involves showing that the argument to Abs_fin_defl in the definition of defl_fun2 is actually a finite deflation. In turn, this requires that the e and p parameters form an ep-pair, and also that the map parameter f preserves finite deflations.

```
lemma cast_defl_fun2:
 assumes "ep_pair e p"
 assumes
  "⋀a b. ⟦finite_deflation a; finite_deflation b⟧ ⟹ finite_deflation (f·a·b)"
 shows "cast·(defl_fun2 e p f·A·B) = e oo f·(cast·A)·(cast·B) oo p"
```

In addition to defl_fun2, we also define a combinator defl_fun1 for use with single-argument type constructors. Using these, we define deflation combinators for all of the type constructors in HOLCF (lifting, product, strict product, strict sum, continuous function space, and all three powerdomains).

6.5.4 Type class of representable domains

We define a *representable* domain as a cpo that can be embedded (via an ep-pair) into the universal domain. Furthermore, we require that the composition of emb and prj must yield an *algebraic* deflation on the universal domain.

```
class domain = pcpo +
 fixes emb :: "'a → udom"
 fixes prj :: "udom → 'a"
```

```
  fixes defl :: "'a itself ⇒ udom defl"
  assumes ep_pair_emb_prj: "ep_pair emb prj"
  assumes cast_DEFL: "cast·(defl TYPE('a)) = emb oo prj"
```

For convenience, we also define the syntax DEFL('a) as shorthand for defl TYPE('a).

Unlike the class rep shown earlier, this definition of class domain is provably a subclass of bifinite. The proof uses lemma obtain_principal_chain from the ideal_completion locale of Chapter 5. Given any algebraic deflation t, there exists a chain of finite deflations whose least upper bound is t. This lemma is applied to obtain a chain of finite deflations whose least upper bound is DEFL('a). We can then compose the elements of this chain with emb and prj to construct an approx-chain on type 'a.

In HOLCF '11, class domain replaces class pcpo as the default sort: All the types most HOLCF users will typically work with must be in class domain. This means that we need class instances for each of the type constructors in HOLCF.

Recall the Haskell Rep class instance for the function type:

```
instance (Rep a, Rep b) => Rep (a -> b) where
  emb = embFun . mapFun prj emb
  prj = mapFun emb prj . prjFun
```

The HOLCF domain class instance for the continuous function space is defined in precisely the same way.

```
instantiation cfun :: (domain, domain) domain
begin
  definition "emb = cfun_emb oo cfun_map·prj·emb"
  definition "prj = cfun_map·emb·prj oo cfun_prj"
  definition "defl (_ :: ('a → 'b) itself) = cfun_defl·DEFL('a)·DEFL('b)"
  instance ...
end
```

The instance proof requires us to show that emb and prj are an ep-pair; this is easy to show using lemmas from Fig. 6.1 like ep_pair_comp and ep_pair_cfun_map.

We must also show that cast·DEFL('a → 'b) = emb oo prj, which follows without too much trouble from cast_cfun_defl and cast_DEFL, using the fact that cfun_map preserves function composition. Class instantiations for the other HOLCF type constructors (lifting, product, strict product, strict sum, and powerdomains) all follow the same pattern.

For base types like unit and 'a lift, for which we do not have corresponding deflation combinators, the class instantiations are a little different. For these types, we can explicitly construct an approx-chain a to establish a bifinite class instance. Then emb and prj can be defined directly as udom_emb a and udom_prj a from the universal domain library. Finally, defl can be defined explicitly as a least upper bound of a chain of finite deflations on udom, based on the finite deflations from the approx-chain a.

6.6 THE DEFINITIONAL DOMAIN PACKAGE

Equipped with formalizations of the universal domain and algebraic deflations, it is now possible to implement the Domain package in a completely definitional way, without generating axioms. The purpose of this section is to describe how the new definitional implementation works—how each "axiom" can now be derived as a theorem.

The definitional Domain package reuses nearly all the code from the axiomatic Domain package. Figure 6.7 contains a copy of the implementation schematic from Chapter 4; recall that the part in the dotted lines is where the axioms are generated. The definitional Domain package is identical to the axiomatic one, except that it uses a drop-in replacement for this part.

The axiomatic Domain package generates two kinds of axioms: First, isomorphism axioms state that the rep and abs functions for each new type are each other's inverses. Second, the reach axiom for each type states a low-level induction rule, asserting that the least upper bound of a chain of take functions is the

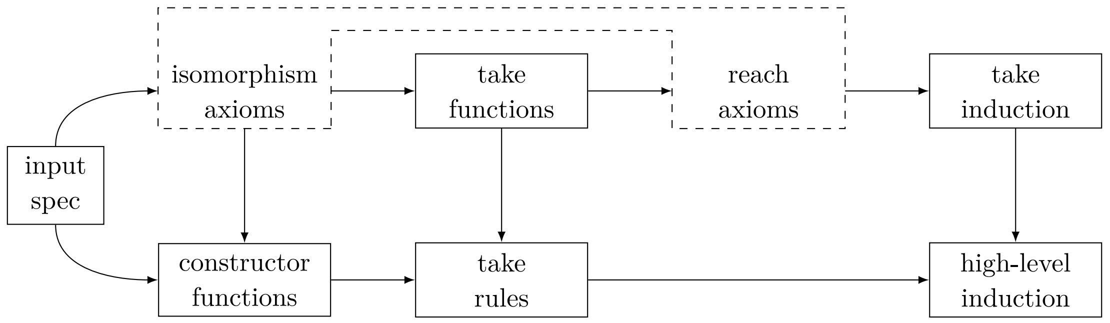


Figure 6.7: Domain package implementation schematic

identity. The remainder of this section will show how these axioms are derived as theorems, using a lazy list datatype as an example to help explain the process.

domain 'a llist = LNil | LCons (**lazy** "'a") (**lazy** "'a llist")

6.6.1 Proving the isomorphism theorems

To construct a domain isomorphism for the lazy list type, information about data constructors is irrelevant; all we need is the domain equation $\mathsf{'a\ llist} \cong \mathsf{one} \oplus (\mathsf{'a}_\bot \otimes \mathsf{'a\ llist}_\bot)$. Solving this domain equation is a three-step process: The definitional Domain package first defines a deflation combinator, then uses the deflation to define the 'a llist type, and finally defines the isomorphism functions.

Defining deflation combinators. The first step is to define a deflation combinator llist_defl :: udom defl → udom defl that models the lazy list type constructor. The deflation combinator is determined by the domain equation that type 'a llist must satisfy:

$$\mathsf{'a\ llist} \cong \mathsf{one} \oplus (\mathsf{'a}_\bot \otimes \mathsf{'a\ llist}_\bot)$$

The definition of llist_defl will refer to the deflation combinator of each type constructor mentioned on the right-hand side of this domain equation. To keep track of

```
lemma [domain_defl_simps]:
  "DEFL('a → 'b) = cfun_defl·DEFL('a)·DEFL('b)"
  "DEFL('a ⊕ 'b) = ssum_defl·DEFL('a)·DEFL('b)
  "DEFL('a ⊗ 'b) = sprod_defl·DEFL('a)·DEFL('b)"
  "DEFL('a × 'b) = prod_defl·DEFL('a)·DEFL('b)"
  "DEFL('a⊥) = u_defl·DEFL('a)"
  "DEFL(('a)♯) = upper_defl·DEFL('a)"
  "DEFL(('a)♭) = lower_defl·DEFL('a)"
  "DEFL(('a)♮) = convex_defl·DEFL('a)"
```

Figure 6.8: Extensible set of rules with the domain_defl_simps attribute

the correspondence between type constructors and deflation combinators, the Domain package maintains an extensible list of theorems with the [domain_defl_simps] attribute. The initial set of these rules is shown in Fig. 6.8.

Using domain_defl_simps, the Domain package produces the recursive specification of llist_defl shown below as llist_defl_unfold. The actual non-recursive definition of llist_defl, using the least fixed point combinator fix, is created using the same machinery employed by the Fixrec package; llist_defl_unfold is then derived from the non-recursive definition as a theorem.

```
theorem llist_defl_unfold:
  "llist_defl·t = ssum_defl·DEFL(one)·(sprod_defl·(u_defl·t)·(u_defl·(llist_defl·t)))"
```

Defining representable domains from deflations. The second step is to use llist_defl to define the actual 'a llist type. In Sections 6.5.3 and 6.5.4, the aim was to build algebraic deflations of type udom defl to correspond with pre-existing cpo types. Now, we need to do the converse: Given a deflation of type udom defl, we must define the corresponding cpo. We can accomplish this with the help of the Cpodef package.

The first step in defining a new cpo type with Cpodef is to determine a subset of values of the old cpo type. We define the defl_set function for this purpose: It produces the image set of any deflation t, defined as the set of fixed points of cast·t.

```
definition defl_set :: "'a defl ⇒ 'a set"
  where "defl_set t = {x. cast·t·x = x}"
```

Now we can define 'a llist as a pointed cpo, isomorphic to the image set of the deflation llist_defl·DEFL('a).

```
pcpodef (open) 'a llist = "defl_set (llist_defl·DEFL('a))"
```

The Cpodef package generates two proof obligations: First, that membership in the given set is admissible, and also that the given set contains ⊥. Both of these properties hold for any application of defl_set.

```
lemma adm_defl_set: "adm (λx. x ∈ defl_set t)"
```

```
lemma defl_set_bottom: "⊥ ∈ defl_set t"
```

The **pcpodef** command only proves that type 'a llist is a pcpo; we still need to provide an instantiation of the domain class. This instantiation requires definitions of emb, prj, and defl, which are defined as follows:

```
instantiation llist :: (domain) domain
begin
  definition "(emb :: 'a llist → udom) ≡ (Λ x. Rep_llist x)"
  definition "(prj :: udom → 'a llist) ≡
      (Λ x. Abs_llist (cast·(llist_defl·DEFL('a))·x))"
  definition "defl ≡ (λ(_ :: 'a llist itself). llist_defl·DEFL('a))"
  instance ...
end
```

The instance proof requires us to show that emb and prj are an ep-pair, and also that the composition emb oo prj is equal to cast·DEFL('a llist). Instead of repeating this proof for each new domain definition, we employ a generic lemma that proves the OFCLASS predicate, in the style of the lemmas used to implement Cpodef.

```
lemma typedef_domain_class:
  fixes Rep :: "'a::pcpo ⇒ udom"
  fixes Abs :: "udom ⇒ 'a::pcpo"
  fixes t :: "udom defl"
  assumes type: "type_definition Rep Abs (defl_set t)"
  assumes below: "below ≡ (λx y. Rep x ⊑ Rep y)"
  assumes emb: "emb ≡ (Λ x. Rep x)"
  assumes prj: "prj ≡ (Λ x. Abs (cast·t·x))"
  assumes defl: "defl ≡ (λ(_ :: 'a itself). t)"
  shows "OFCLASS('a, domain_class)"
```

The assumptions type and below in this rule can be satisfied by theorems generated by Cpodef; the other assumptions are simply the definitions from the domain class instantiation.

After proving the class instance, we can extend domain_defl_simps with a new rule for the lazy list type constructor; this rule follows immediately from the definition of defl for lazy lists.

```
theorem DEFL_llist [domain_defl_simps]: "DEFL('a llist) = llist_defl·DEFL('a)"
```

This entire process of defining a representable domain from a deflation is automated with the Domaindef package, which is built as a thin layer on top of the Cpodef package. Domaindef is called internally by the definitional Domain package, and it is also available as a user-level command. Unlike Cpodef, it is completely automatic; no additional proof obligations are needed.

```
domaindef 'a llist = "llist_defl·DEFL('a)"
```

The above **domaindef** command defines type 'a llist, proves a domain class instance, and adds theorem DEFL_llist to domain_defl_simps.

Constructing isomorphism functions as coercions. The third step is to actually construct the desired domain isomorphism, by defining continuous rep and abs functions and proving that they are each other's inverses.

For any representable domains 'a and 'b, composing emb :: 'a → udom with prj :: udom → 'b yields a *coercion* prj oo emb :: 'a → 'b. If DEFL('a) ⊑ DEFL('b), then the coercions from 'a to 'b and back form an ep-pair. If DEFL('a) = DEFL('b), then the coercions from 'a to 'b and back form a continuous isomorphism. The Domain package uses this fact to define the rep and abs functions for each new domain, and show that they form a continuous isomorphism.

```
definition llist_rep :: "'a llist → one ⊕ ('a⊥ ⊗ 'a llist⊥)"
  where "llist_rep ≡ prj oo emb"

definition llist_abs :: "one ⊕ ('a⊥ ⊗ 'a llist⊥) → 'a llist"
  where "llist_abs ≡ prj oo emb"
```

Before we can prove that llist_rep and llist_abs are a continuous isomorphism, we must prove that the types they coerce between are represented by the same deflation. Theorem DEFL_eq_llist is easily proved using domain_defl_simps together with llist_defl_unfold.

```
theorem DEFL_eq_llist: "DEFL('a llist) = DEFL(one ⊕ ('a⊥ ⊗ 'a llist⊥))"
```

Using DEFL_eq_llist, the definitional Domain package can *prove* that llist_abs and llist_rep are an isomorphism—in the axiomatic Domain package, llist.abs_iso and llist.rep_iso would have been declared as axioms.

```
theorem llist.abs_iso: "llist_rep·(llist_abs·x) = x"

theorem llist.rep_iso: "llist_abs·(llist_rep·y) = y"
```

Theorem llist.abs_iso is proved by instantiating lemma domain_abs_iso from the library with DEFL_eq_llist and the definitions of llist_abs and llist_rep.

```
lemma domain_abs_iso:
  fixes abs :: "'a → 'b" and rep :: "'b → 'a"
  assumes DEFL: "DEFL('b) = DEFL('a)"
  assumes abs_def: "abs ≡ prj oo emb"
  assumes rep_def: "rep ≡ prj oo emb"
  shows "rep·(abs·x) = x"
```

```
theorem llist.take_def:
  "llist_take ≡ (λn. iterate n·
    (Λ g. llist_abs oo ssum_map·ID·(sprod_map·ID·(u_map·g)) oo llist_rep)·⊥)"

theorem llist.take_0: "llist_take 0 = ⊥"

theorem llist.take_Suc: "llist_take (Suc n) =
  llist_abs oo ssum_map·ID·(sprod_map·ID·(u_map·(llist_take n))) oo llist_rep"

theorem llist.chain_take: "chain llist_take"

theorem llist.deflation_take: "deflation (llist_take n)"
```

Figure 6.9: Definition and basic properties of llist_take function

Theorem llist_rep_iso is proved using a similar rule, domain_rep_iso, which concludes $\mathsf{abs}\cdot(\mathsf{rep}\cdot\mathsf{y}) = \mathsf{y}$ from the same assumptions.

6.6.2 Proving the reach lemma

After the new definitional Domain package code constructs the isomorphism, the take-function component (described in Chapter 4) defines llist_take, and proves some theorems (listed in Fig. 6.9). Now the definitional Domain package needs to prove the reach lemma, which states that the least upper bound of the chain llist_take is the identity function.

As an integrated part of the process of deriving the reach lemma, the definitional Domain package also performs the necessary steps to allow indirect recursion with llist in later domain definitions. This involves defining a map function llist_map, proving a few theorems about it, and adding those theorems to the appropriate databases.

The overall process can be broken down into three main steps: First, define the map function. Second, prove the identity law for the map function, by establishing a relationship between the map function and the deflation combinator. Third, prove the reach lemma by relating the take function to the map function.

Defining the map function. The Domain package defines the map function llist_map of type ('a → 'a) → 'a llist → 'a llist.[1] It generates a recursive specification from a combination of other map functions, based on the structure of the type $\text{one} \oplus (\text{'a}_{\bot} \otimes \text{'a llist}_{\bot})$. As with the deflation combinator, the actual fixed point definition is handled by Fixrec-style machinery.

```
theorem llist_map_unfold:
  "llist_map·f = llist_abs oo
    ssum_map·ID·(sprod_map·(u_map·f)·(u_map·(llist_map·f))) oo llist_rep"
```

At this point the Domain package proves one of the rules needed for later indirect-recursive definitions, which states that llist_map preserves deflations. Theorem deflation_llist_map is proved by fixed point induction, using the pre-existing domain_deflation rules; it is then added to the domain_deflation rule database.

```
theorem deflation_llist_map [domain_deflation]:
  "deflation f ⟹ deflation (llist_map·f)"
```

Proving the identity law. The other property that we must prove about llist_map is the identity law, llist_map·ID = ID. To prove this property, the Domain package exploits the similarities in the recursive definitions of llist_map and the llist_defl. The relationship between the two is formalized in a binary relation called isodefl, which states that the given function f is "isomorphic" in some sense to the algebraic deflation t.

```
definition isodefl :: "('a::domain → 'a) ⇒ udom defl ⇒ bool"
  where "isodefl f t = (cast·t = emb oo f oo prj)"
```

[1] The most general type of llist_map is actually ('a → 'b) → 'a llist → 'b llist. However, the current implementation uses the more restrictive type scheme because it works without modification for contravariant types. Generalizing the types of map functions is planned as future work.

```
lemma [domain_isodefl]:
  "isodefl (ID :: 'a → 'a) DEFL('a)"
  "⟦isodefl f1 t1; isodefl f2 t2⟧ ⟹ isodefl (cfun_map·f1·f2) (cfun_defl·t1·t2)"
  "⟦isodefl f1 t1; isodefl f2 t2⟧ ⟹ isodefl (ssum_map·f1·f2) (ssum_defl·t1·t2)"
  "⟦isodefl f1 t1; isodefl f2 t2⟧ ⟹ isodefl (sprod_map·f1·f2) (sprod_defl·t1·t2)"
  "⟦isodefl f1 t1; isodefl f2 t2⟧ ⟹ isodefl (prod_map·f1·f2) (prod_defl·t1·t2)"
  "isodefl f t ⟹ isodefl (u_map·f) (u_defl·t)"
  "isodefl f t ⟹ isodefl (upper_map·f) (upper_defl·t)"
  "isodefl f t ⟹ isodefl (lower_map·f) (lower_defl·t)"
  "isodefl f t ⟹ isodefl (convex_map·f) (convex_defl·t)"
```

Figure 6.10: Extensible set of rules with the domain_isodefl attribute

The Domain package maintains a database of theorems relating map functions to their corresponding deflation combinators, using the attribute [domain_isodefl]. The initial contents of this database are shown in Fig. 6.10.

Now the Domain package must prove a similar domain_isodefl rule for the lazy list type:

```
theorem isodefl_llist [domain_isodefl]:
  "isodefl f t ⟹ isodefl (llist_map·f) (llist_defl·t)"
```

The proof proceeds by a form of parallel fixed point induction: After unfolding the definitions of llist_map and llist_defl to reveal the underlying fixed point combinators, rule parallel_fix_ind can be applied.

```
lemma parallel_fix_ind:
  "⟦adm (λx. P (fst x) (snd x)); P ⊥ ⊥; ⋀x y. P x y ⟹ P (F·x) (G·y)⟧
    ⟹ P (fix·F) (fix·G)"
```

After discharging the admissibility check and the base case isodefl ⊥ ⊥, the final subgoal is nontrivial:

```
goal (1 subgoal):
 1. ⋀x y. ⟦isodefl f t; isodefl x y⟧
```

```
⟹ isodefl
    (llist_abs oo
     ssum_map·ID·(sprod_map·(u_map·f)·(u_map·x)) oo llist_rep)
    (ssum_defl·DEFL(one)·(sprod_defl·(u_defl·t)·(u_defl·y)))
```

This last subgoal is solved by repeatedly applying rules from the domain_isodefl database, along with one extra rule to handle the occurrences of llist_abs and llist_rep:

```
theorem llist.isodefl_abs_rep:
  "isodefl f t ⟹ isodefl (llist_abs oo f oo llist_rep) t"
```

A similar theorem holds for any rep and abs functions defined as coercions between isomorphic types.

Once theorem isodefl_llist has been proven, the Domain package uses lemmas isodefl_DEFL_imp_ID and DEFL_llist to derive the identity law for llist_map as a corollary.

```
lemma isodefl_DEFL_imp_ID: "isodefl f DEFL('a) ⟹ f = ID"
```

```
theorem llist_map_ID [domain_map_ID]: "llist_map·ID = ID"
```

Once proved, llist_map_ID is added to the domain_map_ID database (introduced in Chapter 4) for use in later domain definitions.

Relating the map and take functions. In the final step, the reach lemma can be derived from the identity law of llist_map, by taking advantage of similarities in the definitions of llist_take and llist_map.

```
theorem llist.lub_take: "(⨆n. llist_take n) = ID"
```

The proof begins by applying transitivity with llist_map_ID, yielding the subgoal (⨆n. llist_take n) = llist_map·ID. After unfolding the definitions of llist_take, llist_map, and the fixed point combinator fix, both sides of the equality have the form (⨆n. iterate·f·⊥). Rewriting with domain_map_ID rules can finish the proof.

6.6.3 User-visible changes

In HOLCF '11, the Domain package operates in definitional mode by default. For backward compatibility, the axiomatic mode is still available using the **domain** (unsafe) command. The primary user-visible difference between the two modes is which type classes they use: In definitional mode, type parameters and other constructor argument types must be in class domain; newly-defined datatypes are also made instances of the domain class. In contrast, the axiomatic mode uses the pcpo class throughout. Even with this change, most HOLCF user theories work without modification, because the default sort has also changed from pcpo to domain.

The other user-visible change is the new support for indirect recursion. After defining type 'a llist with the definitional Domain package, users can define other datatypes using indirect recursion with llist, such as this datatype of trees:

domain 'a tree = Leaf (**lazy** "'a") | Node (**lazy** "'a tree llist")

In axiomatic mode, however, the Domain package does not generate map functions, and does not configure indirect recursion to work with new datatypes.

6.7 UNPOINTED PREDOMAINS

Prior to HOLCF '11, pcpo has always been the default sort; any type variables mentioned in HOLCF theories were assumed to be in class pcpo by default. However, the class cpo, which is a superclass of pcpo that does not require a bottom element, is also useful in some cases. Various theory developments based on HOLCF '99 have made use of the cpo class [MNOS99, Mül98]. With HOLCF '11 switching from pcpo to the domain class, what should be done with these theories that use class cpo? There is a need for an unpointed variant of class domain in HOLCF '11.

To fill this need, we introduce a class of *predomains*. A predomain is a cpo that, when lifted, becomes a representable domain. More precisely, while 'a::domain means that type 'a can be embedded into udom, 'a::predomain means that type 'a$_\perp$ can be embedded into udom$_\perp$.

```
class predomain_syn = cpo +
  fixes liftemb :: "'a⊥ → udom⊥"
  fixes liftprj :: "udom⊥ → 'a⊥"
  fixes liftdefl :: "'a itself ⇒ (udom⊥) defl"

class predomain = predomain_syn +
  assumes predomain_ep: "ep_pair liftemb liftprj"
  assumes cast_liftdefl: "cast·(liftdefl TYPE('a)) = liftemb oo liftprj"
```

We also define LIFTDEFL('a) as a convenient abbreviation for liftdefl TYPE('a).

Shortly we will redefine the domain type class to make it into a subclass of predomain. But in order to do that, we need a function for creating deflations on udom$_\perp$ out of deflations on udom.

```
definition liftdefl_of :: "udom defl → (udom⊥) defl"
  where "liftdefl_of = defl_fun1 ID ID u_map"

lemma cast_liftdefl_of: "cast·(liftdefl_of·t) = u_map·(cast·t)"
```

We now extend the domain type class by adding predomain_syn as a superclass, along with a few class assumptions stating that liftemb, liftprj, and liftdefl are defined in a standard way.

```
class domain = predomain_syn + pcpo +
  fixes emb :: "'a → udom"
  fixes prj :: "udom → 'a"
  fixes defl :: "'a itself ⇒ udom defl"
  assumes ep_pair_emb_prj: "ep_pair emb prj"
  assumes cast_DEFL: "cast·(defl TYPE('a)) = emb oo prj"
  assumes liftemb_eq: "liftemb = u_map·emb"
  assumes liftprj_eq: "liftprj = u_map·prj"
  assumes liftdefl_eq: "liftdefl TYPE('a) = liftdefl_of·(defl TYPE('a))"
```

It is then a simple matter to prove the subclass relationship domain ⊆ predomain, by showing that the definitions of liftemb, liftprj, and liftdefl specified by the domain class satisfy the predomain axioms.

Domain class instance for lifted cpo. Now we define a new domain class instance for the lifted cpo type $'a_\bot$, so that the type argument 'a only needs to be a predomain. To implement this class instance, we need a new variant of the deflation combinator u_defl whose argument type is $(\text{udom}_\bot)$ defl instead of udom defl.

```
definition u_liftdefl :: "(udom⊥) defl → udom defl"
  where "u_liftdefl = defl_fun1 u_emb u_prj ID"
```

```
lemma cast_u_liftdefl: "cast·(u_liftdefl·t) = u_emb oo cast·t oo u_prj"
```

Here u_emb :: $\text{udom}_\bot \rightarrow$ udom and u_prj :: udom $\rightarrow \text{udom}_\bot$ are the ep-pair provided by the universal domain library—the same functions are used to define u_defl. The two deflation combinators are related by the following theorem:

```
lemma u_liftdefl_liftdefl_of: "u_liftdefl·(liftdefl_of·t) = u_defl·t"
```

This means that although $\text{DEFL}('a_\bot) = \text{u\_liftdefl}\cdot\text{LIFTDEFL}('a)$ by definition, it is also still equal to u_defl·DEFL('a) for any 'a in class domain, just as it was defined before.

```
instantiation u :: (predomain) domain
begin
  definition "(emb :: 'a⊥ → udom) = u_emb oo liftemb"
  definition "(prj :: udom → 'a⊥) = liftprj oo u_prj"
  definition "defl (_ :: ('a⊥) itself) = u_liftdefl·LIFTDEFL('a)"
  ...
end
```

The functions liftemb, liftprj, and liftdefl are all defined exactly as required by the domain class axioms. The other domain class axioms about emb, prj, and defl follow from the predomain class axioms on type 'a.

Predomain class instance for cartesian product. The goal here is to create a predomain instance for products, such that the product of two predomains is again a predomain. To do this, we must create an ep-pair from type $(\mathsf{'a} \times \mathsf{'b})_\perp$ into $\mathsf{udom}_\perp$. To define the embedding and projection, we make use of an isomorphism between type $(\mathsf{'a} \times \mathsf{'b})_\perp$ and the strict product $\mathsf{'a}_\perp \otimes \mathsf{'b}_\perp$.

```
definition encode_prod_u :: "('a × 'b)⊥ → 'a⊥ ⊗ 'b⊥"
  where "encode_prod_u = (Λ(up·(x, y)). (:up·x, up·y:))"

definition decode_prod_u :: "'a⊥ ⊗ 'b⊥ → ('a × 'b)⊥"
  where "decode_prod_u = (Λ(:up·x, up·y:). up·(x, y))"
```

The embedding can now be done in multiple steps: Starting with a value of type $(\mathsf{'a} \times \mathsf{'b})_\perp$, first apply encode_prod_u to get a strict pair of type $\mathsf{'a}_\perp \otimes \mathsf{'b}_\perp$. Second, map liftemb over each component of the strict pair to get another pair of type $\mathsf{udom}_\perp \otimes \mathsf{udom}_\perp$. Third, apply decode_prod_u to convert this to type $(\mathsf{udom} \times \mathsf{udom})_\perp$. Finally, map the embedding function for type udom × udom over the lifted cpo to get a value of type $\mathsf{udom}_\perp$. The projection uses the same process in reverse. We then define the deflation combinator prod_liftdefl to correspond with the composition of these embedding and projection functions.

```
definition prod_liftdefl :: "(udom⊥) defl → (udom⊥) defl → (udom⊥) defl"
  where "prod_liftdefl = defl_fun2 (u_map·prod_emb oo decode_prod_u)
    (encode_prod_u oo u_map·prod_prj) sprod_map"

lemma cast_prod_liftdefl:
  "cast·(prod_liftdefl·a·b) = (u_map·prod_emb oo decode_prod_u) oo
    sprod_map·(cast·a)·(cast·b) oo (encode_prod_u oo u_map·prod_prj)"

instantiation prod :: (predomain, predomain) predomain
begin
  definition "liftemb = (u_map·prod_emb oo decode_prod_u)
      oo (sprod_map·liftemb·liftemb oo encode_prod_u)"
  definition "liftprj = (decode_prod_u oo sprod_map·liftprj·liftprj)
      oo (encode_prod_u oo u_map·prod_prj)"
  definition "liftdefl (_ :: ('a × 'b) itself) =
      prod_liftdefl·LIFTDEFL('a)·LIFTDEFL('b)"
  instance ...
end
```

Although the definitions may look complex, the predomain instance proof is still relatively simple. Showing ep_pair liftemb liftprj merely requires a few extra applications of rules from Fig. 6.1.

```
instantiation prod :: (domain, domain) domain
begin
  definition "(emb :: 'a × 'b → udom) = prod_emb oo prod_map·emb·emb"
  definition "(prj :: udom → 'a × 'b) = prod_map·prj·prj oo prod_prj"
  definition "defl (_ :: ('a × 'b) itself) = prod_defl·DEFL('a)·DEFL('b)"
  instance ...
end
```

On the other hand, the domain instance proof requires more work than one might expect. The reason is that unlike any other domain class instantiation, the product type does not have liftemb, liftprj, and liftdefl defined exactly as specified by the domain class axioms. Fortunately, liftemb = u_map·emb and liftprj = u_map·prj can be proved without too much trouble by case analysis on their inputs. Also, LIFTDEFL('a × 'b) = liftdefl_of·DEFL('a × 'b) can be proved using the injectivity of cast, by showing that cast·LIFTDEFL('a × 'b) = cast·(liftdefl_of·DEFL('a × 'b)).

Other class instances. Other predomain class instances can be defined in a similar way to the product type. For example, with the Isabelle/HOL disjoint sum type, an isomorphism can be defined between type ('a + 'b)$_\perp$ and the strict sum 'a$_\perp$ ⊕ 'b$_\perp$. This can be used to show that the disjoint sum of two predomains is again a predomain.

Another isomorphism that was noted back in Chapter 2 is between type 'a discr$_\perp$ and 'a lift. This is used to make 'a discr an instance of class predomain, for any countable type 'a.

The last new class instance of note is for the continuous function type. To show that type 'a → 'b is in class pcpo, 'b must be a pcpo, but 'a only needs to be in class cpo. How can we get a similar instance for the domain and predomain classes?

In order to prove this class instance, we can define a type 'a →! 'b, which is the *strict function space* from 'a to 'b.

```
pcpodef (open) ('a, 'b) sfun (infixr "→!" 0) = "{f :: 'a → 'b. f·⊥ = ⊥}"
```

For an unpointed 'a and pointed 'b, type 'a → 'b is isomorphic to the strict function type $\text{'a}_{\perp}$ →! 'b. This can then be used to obtain the desired class instance for continuous functions.

Domain package support for predomains. Some minor changes to the Domain package were necessary to add support for predomains. Two new features were implemented. First, lazy constructor arguments are now permitted to be in class predomain. (Strict constructor arguments are still required to be in class domain.) For example, we can define the following type of lazy lists of natural numbers, where the elements have the unpointed type nat discr:

```
domain natlist = nil | cons (lazy "nat discr") (lazy "natlist")
```

The second feature is that datatypes can now have type parameters in class predomain. For example, we can define a variation of the lazy list datatype that allows unpointed predomain element types:

```
domain ('a::predomain) ulist = unil | ucons (lazy "'a") (lazy "'a ulist")
```

Note that the constructor argument 'a is lazy, as it must be for the definition to be accepted.

To support these new features, the part of the Domain package that defines deflation combinators had to be updated. For domains with unpointed type parameters, the type of the deflation combinator is different: For example, llist_defl had type udom defl → udom defl, while ulist_defl has type $(\text{udom}_{\perp})$ defl → udom defl. Because the domain_defl_simps rules are used to generate deflation combinator definitions, we must also declare a few more rules with this attribute (Fig. 6.11).

```
lemma [domain_defl_simps]:
  "DEFL('a⊥) = u_liftdefl·LIFTDEFL('a)"
  "LIFTDEFL('a::domain) = liftdefl_of·DEFL('a)"
  "u_liftdefl·(liftdefl_of·t) = u_defl·t"
  "LIFTDEFL('a × 'b) = prod_liftdefl·LIFTDEFL('a)·LIFTDEFL('b)"
  "DEFL(('a::predomain) → 'b) = DEFL('a⊥ →! 'b)"
```

Figure 6.11: Additional domain_defl_simps rules for predomains

```
lemma [domain_isodefl]:
  "isodefl' (ID :: 'a → 'a) LIFTDEFL('a::predomain)"
  "isodefl f t ⟹ isodefl' f (liftdefl_of·t)"
  "isodefl' f t ⟹ isodefl (u_map·f) (u_liftdefl·t)"
  "⟦isodefl' f1 t1; isodefl' f2 t2⟧ ⟹ isodefl' (prod_map·f1·f2) (prod_liftdefl·t1·t2)"
```

Figure 6.12: Additional domain_isodefl rules for predomains

```
theorem ulist_defl_unfold: "ulist_defl·t =
  ssum_defl·DEFL(one)·(sprod_defl·(u_liftdefl·t)·(u_defl·(ulist_defl·t)))"
```

The DOMAINDEF package also needs modification to handle the changed definition of the domain class. Lemma typedef_domain_class gets three extra assumptions, corresponding to the class axioms liftemb_eq, liftprj_eq, and liftdefl_eq; DOMAINDEF defines these constants accordingly for each instance.

For the proofs of the identity law for map functions, we must define a variant of the isodefl relation for use with the predomain class. We also extend the initial set of domain_isodefl rules with a few new ones about isodefl', shown in Fig. 6.12.

```
definition isodefl' :: "('a::predomain → 'a) ⇒ (udom⊥) defl ⇒ bool"
  where "isodefl' f t ⟷ cast·t = liftemb oo u_map·f oo liftprj"
```

The domain_isodefl rules for new domains must mention this variant if they have any predomain type parameters. For example, consider the 'a ulist type:

```
theorem isodefl_ulist [domain_isodefl]:
  "isodefl' f t ⟹ isodefl (ulist_map·f) (ulist_defl·t)"
```

Perhaps surprisingly, these few changes listed here are the only ones necessary to support the new predomain features. All of the other proof scripts in the rest of the Domain package continue to work without modification in the presence of predomains.

6.8 RELATED WORK AND CONCLUSION

An early example of the purely definitional approach to defining datatypes is described by Melham, in the context of the HOL theorem prover [Mel89]. Melham defines a type $(\alpha)\,\mathit{Tree}$ of labelled trees, from which other recursive types are defined as subsets. The design is similar in spirit to the one presented in this chapter—types are modeled as values, and abstract properties that characterize each datatype are proved as theorems. The main differences are that Melham uses ordinary types instead of bifinite domains, and ordinary subsets instead of deflations.

The Isabelle/HOL datatype package uses a design very similar to the HOL system. The type α *node*, which was originally used for defining recursive types in Isabelle/HOL, was introduced by Paulson [Pau97]; it is quite similar to the HOL system's $(\alpha)\,\mathit{Tree}$ type. E. Gunter later extended the labelled tree type of HOL to support datatypes with arbitrary branching [Gun94]. Berghofer and Wenzel used a similarly extended type to implement the current version of Isabelle's datatype package [BW99].

Agerholm used a variation of Melham's labelled trees to define lazy lists and other recursive domains in the HOL-CPO system [Age94]. Agerholm's cpo of infinite trees can represent arbitrary polynomial datatypes as subsets; however, negative recursion is not supported.

Recent work by Benton, et al. uses the colimit construction to define recursive domains in Coq [BKV09]. Like the universal domain described in this chapter, their technique can handle both positive and negative recursion. Using colimits avoids the need for a universal domain, but it requires a logic with dependent types; the construction will not work in ordinary higher-order logic.

On the theoretical side, various publications by C. Gunter [Gun85, Gun87, Gun92] were the primary sources of ideas for the HOLCF universal domain construction. The construction of the sequence of increments in Section 6.4 is just as described by Gunter [Gun87, §5]. However, the use of ideal completion is original—Gunter defines the universal domain using a colimit construction instead. Given a cpo D, Gunter defines a type D^+ that can embed any increment from D to D'. The universal domain is then defined as a solution to the domain equation $D = D^+$. The construction of D^+ is similar to our basis datatype B, except that it is non-recursive and does not include serial numbers.

Another universal domain with similar features has been described by D. Scott [GS90, Sco08]. The domain U is defined by ideal completion, where the basis is a countable free boolean algebra minus its top element. It is proved to be a universal domain for the class of *bounded complete* algebraic cpos (also commonly known as *Scott domains*) which is a subclass of the bifinite cpos. Compared to the universal bifinite domain udom, its basis has a much simpler definition, and would have been significantly easier to formalize; however, bounded completeness is not preserved by the convex powerdomain.

While the literature only claims that the domain U can represent bounded complete cpos, it is actually possible that it could represent arbitrary *bifinite* cpos as well. The construction described in this chapter is based on encoding increments of posets, where the new element $\langle i, a, S\rangle$ with serial number i is inserted above the element a and below each element in the finite set S. In the free boolean algebra with generators $(x_i)_{i\in\omega}$, we could encode the value $\langle i, a, S\rangle$ as $(x_i \sqcap a) \sqcup (\neg x_i \sqcap (\bigsqcap S))$. As long as each inserted value uses a distinct serial number, this encoding appears to satisfy the appropriate ordering relations. Exploring this idea in more detail is reserved for future work.

Earlier versions of portions of this chapter have been published previously [Huf09a]. The formalization of the universal domain in HOLCF '11 differs sightly from that earlier presentation in its treatment of the chain of approx functions: Previously, approx was an overloaded function of the bifinite type class, and udom_emb and udom_prj were polymorphic functions in the same type class. In contrast, the new version uses a locale to fix the chain of approx functions.

In summary, the HOLCF '11 universal domain library provides the basic infrastructure upon which the new Domain package can construct general recursive datatypes in a purely definitional way. It provides a type udom, along with the means to construct an ep-pair into udom from any other bifinite cpo. Such ep-pairs are a prerequisite for building deflation combinators that represent type constructors. In turn, the Domain package now uses deflation combinators to define recursive datatypes, proving the type isomorphism and induction rules without generating axioms.

Chapter 7

CASE STUDY AND CONCLUSION: VERIFYING MONADS

7.1 INTRODUCTION

The primary claim of this thesis is that HOLCF '11 offers a superior environment for program verification—specifically, we claim that HOLCF '11 provides an unprecented combination of *expressiveness*, *automation*, and *confidence*. Expressiveness means that users can accurately and concisely specify the datatypes, functions, and properties that they want to reason about. The new and improved definition packages in HOLCF '11 make it easy to translate a wider variety of recursive datatype and function definitions than ever before. Automation means that users can avoid expending effort on trivial proof details, and focus on the interesting parts of more difficult proofs. Theorems that are straightforward to prove on paper (and also some that are not so easy on paper) have almost completely automatic proofs in HOLCF '11. The new proof automation also helps users to prove large, complex theorems with minimal effort. Confidence means that there is a strong argument for believing in the correctness of the system—HOLCF '11 provides confidence by adhering to a purely definitional approach, and avoiding new axioms.

The goal of this chapter is to provide evidence for these claims. The level of confidence has already been established in the previous chapters, which describe the purely definitional implementation of HOLCF '11. To measure the expressiveness and automation available in HOLCF '11, we consider case studies where we prove properties about specific functional programs. Finally, to support the claim that the combination of these qualities in HOLCF '11 is *unprecedented*, we compare HOLCF '11 with some earlier approaches to program verification, evaluating them along the axes of expressiveness, automation, and confidence.

For the case studies in this chapter, we examine some monad types of the kind often used in Haskell programming. Recall that monads are typically used to implement computations with side-effects, such as exceptions or mutable state—different monad types provide different kinds of side-effects. Every monad is equipped with a *return* operation, which returns a value without side-effects; and a *bind* operation, which sequences computations by feeding the result of one computation into the next. These operations are expected to satisfy a set of monad laws. Also, individual monads may have other operations that are expected to satisfy additional laws. For example, a mutable-state monad would have *read* and *write* operations that should satisfy some simple properties.

The example monads used in this chapter range from simple to complex: First we have the basic type of lazy lists, which is ubiquitous in Haskell programming. Later on we consider a much more complex monad with multiple kinds of side-effects, that is useful for modeling concurrency. Its definition is built up from simpler monads in stages, using *monad transformers* [Mog89, LHJ95]. The definitions and proofs for lazy lists serve primarily to show how well HOLCF '11 handles easy verification tasks, while the concurrency monad is intended to test the full extent of HOLCF '11's capabilities.

In addition to verifying the monad laws for each type, we also define and verify some other type-specific operations. For lazy lists, we formalize the Haskell functions `repeat`, which generates infinite lists; and `zipWith`, which applies a function pointwise to two lists. For the concurrency monad, we formalize an operation that nondeterministically interleaves two computations. Each of these operations satisfies a set of laws that comes from the theory of *applicative functors* [MP08].

Contributions. The case studies here primarily serve to demonstrate the tools described in the earlier chapters, but they introduce some new technical contributions as well. Some new proof techniques have been developed to help automate some of the proofs presented in this chapter:

- Parallel fixed point induction using depth parameters (§7.2.5)
- Principle of *map-induction* for indirect-recursive datatypes (§7.3.6)

Overview. The remainder of this chapter is organized as follows: Section 7.2 covers the lazy list monad case study. After defining the datatype and list operations (§7.2.1), we verify the monad laws (§7.2.2). Next we consider the applicative functor instance with `repeat` and `zipWith` (§7.2.3) and their correctness proofs (§7.2.4). One law in particular requires more advanced coinductive proof techniques; we evaluate some standard methods, and compare them with a new technique using induction over depth parameters (§7.2.5).

Section 7.3 discusses the development of the concurrency monad. The concurrency monad is built up using a sequence of standard monads and monad transformers (§7.3.1–7.3.4), ultimately being defined by the Domain package (§7.3.5). After developing the *map-induction* principle for reasoning about the concurrency monad (§7.3.6), we verify the monad operations (§7.3.7) and the nondeterministic interleaving operator (§7.3.8).

Section 7.5 contains a survey of related work, where we compare the attributes of HOLCF '11 with those of various earlier systems and approaches to program verification. Finally, Sec. 7.6 gives a summary and some closing remarks.

7.2 THE LAZY LIST MONAD

In this section, we formalize a lazy list type in HOLCF, which models the standard Haskell list type. We also define the functor and monad operations on the lazy list type, corresponding to the standard Haskell definitions of `fmap`, `return` and `(>>=)`

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

        fmap id v = v               -- Identity
fmap g (fmap h v) = fmap (g . h) v  -- Composition
```

Figure 7.1: Haskell class `Functor`, with functor laws

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b

 (return x >>= g) = g x                         -- Left unit
   (v >>= return) = v                           -- Right unit
((v >>= g) >>= h) = (v >>= (\x -> g x >>= h))   -- Associativity
```

Figure 7.2: Haskell class `Monad`, with monad laws

for lists, and prove the standard laws about them. The definitions and laws for the functor and monad classes are shown in Figs. 7.1 and 7.2; the class instances for lists are given in Fig. 7.3.

HOLCF '11 aims to make easy proofs automatic, and hard proofs possible. These properties of Haskell list operations are examples of easy proofs—the goal of this section is to illustrate how simple it is to formalize the definitions in HOLCF, and how easy and automated the proofs can be.

7.2.1 Datatype and function definitions

The first step in formalizing the Haskell list monad in HOLCF is to define the type using the Domain package. We use the name llist for lazy lists in HOLCF, to avoid a clash with the existing Isabelle/HOL list datatype.

domain 'a llist = LNil | LCons (**lazy** "'a") (**lazy** "'a llist")

```
data [a] = [] | a : [a]

instance Functor [] where
  fmap g [] = []
  fmap g (x : xs) = g x : fmap g xs

(++) :: [a] -> [a] -> [a]
[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)

instance Monad [] where
  return x = [x]
  []       >>= k = []
  (x : xs) >>= k = k x ++ (xs >>= k)
```

Figure 7.3: Haskell `Functor` and `Monad` instances for lazy lists

Among the many theorems generated by this call to the DOMAIN package, one of the most important is llist.induct: It is used in every proof that involves induction over lazy lists.

theorem llist.induct:
"⟦adm P; P ⊥; P LNil; ⋀x xs. P xs ⟹ P (LCons·x·xs)⟧ ⟹ P ys"

We formalize the Haskell list functions `fmap`, `return`, `(++)`, and `(>>=)` in HOLCF as mapL, unitL, appendL, and bindL, respectively (see Fig. 7.4). Each of them is defined using FIXREC, except for unitL; because unitL does not need pattern matching or recursion, a simple **definition** suffices. In addition to the defining equations, we also generate strictness rules for each function. Each strictness rule is proved by a single application of the fixrec_simp method (introduced in Chapter 3).

```
fixrec mapL :: "('a → 'b) → 'a llist → 'b llist"
  where "mapL·f·LNil = LNil"
  | "mapL·f·(LCons·x·xs) = LCons·(f·x)·(mapL·f·xs)"

definition unitL :: "'a → 'a llist"
  where "unitL = (Λ x. LCons·x·LNil)"

fixrec appendL :: "'a llist → 'a llist → 'a llist"
  where "appendL·LNil·ys = ys"
  | "appendL·(LCons·x·xs)·ys = LCons·x·(appendL·xs·ys)"

fixrec bindL :: "'a llist → ('a → 'b llist) → 'b llist"
  where "bindL·LNil·f = LNil"
  | "bindL·(LCons·x·xs)·f = appendL·(f·x)·(bindL·xs·f)"

lemma mapL_strict [simp]: "mapL·f·⊥ = ⊥"
  by fixrec_simp

lemma appendL_strict [simp]: "appendL·⊥·ys = ⊥"
  by fixrec_simp

lemma bindL_strict [simp]: "bindL·⊥·f = ⊥"
  by fixrec_simp
```

Figure 7.4: HOLCF formalization of functor and monad operations for lazy lists

7.2.2 Verifying the functor and monad laws

With all the operations defined and the rewrite rules added to the simplifier, we can proceed to prove the functor and monad laws. The proofs of both functor laws for lazy lists are completely automatic: Just apply induction followed by simplification.

```
lemma mapL_ID: "mapL·ID·xs = xs"
  by (induct xs, simp_all)

lemma mapL_mapL: "mapL·f·(mapL·g·xs) = mapL·(f oo g)·xs"
  by (induct xs, simp_all)
```

The right unit law for the lazy list monad has a similar automatic proof, as long as we tell the simplifier to unfold the definition of unitL.

```
lemma bindL_unitL_right: "bindL·xs·unitL = xs"
  by (induct xs, simp_all add: unitL_def)
```

The proofs of the other two monad laws are slightly more difficult. While they still have a high level of automation, each proof requires one or more lemmas. We will consider the left unit law first.

The left unit law for the lazy list monad requires that bindL·(unitL·x)·g = g·x. If we unfold the definition of unitL and simplify, we are left with the subgoal appendL·(g·x)·LNil = g·x. To proceed, we must back up and prove a lemma saying that appending LNil on the right leaves a list unchanged. With the help of lemma appendL_LNil_right, the left unit law can then be proved automatically.

```
lemma appendL_LNil_right: "appendL·xs·LNil = xs"
  by (induct xs, simp_all)

lemma bindL_unitL: "bindL·(unitL·x)·g = g·x"
  by (simp add: unitL_def appendL_LNil_right)
```

Finally we consider the associativity law, which asserts that bindL·(bindL·xs·g)·h = bindL·xs·(Λ x. bindL·(g·x)·h). If we perform induction on xs, the simplifier can automatically solve all subgoals but one: In the LCons case, the left hand side bindL·(bindL·(LCons·x·xs)·g)·h reduces to bindL·(appendL·(g·x)·(bindL·xs·g))·h, but then we get stuck: We need to prove a lemma to show that bindL distributes over appendL.

We can try to prove bindL·(appendL·xs·ys)·g = appendL·(bindL·xs·g)·(bindL·ys·g) as a lemma, by induction on xs. As before, simplification discharges everything but the LCons·x·xs case: After simplifying and rewriting with the inductive hypothesis, the left-hand side reduces to appendL·(g·x)·(appendL·(bindL·xs·g)·(bindL·ys·g)). The right-hand side is similar, but has the appends grouped the other way: appendL·(appendL·(g·x)·(bindL·xs·g))·(bindL·ys·g). This suggests yet another lemma: We must prove that appendL is associative.

At last, the associativity of appendL can be proved directly by induction on xs, without needing any more lemmas. After appendL_appendL, we can now give fully automatic proofs for bindL_appendL and bindL_bindL, where each theorem uses the previous one as a rewrite rule.

```
lemma appendL_appendL:
    "appendL·(appendL·xs·ys)·zs = appendL·xs·(appendL·ys·zs)"
  by (induct xs, simp_all)

lemma bindL_appendL:
    "bindL·(appendL·xs·ys)·g = appendL·(bindL·xs·g)·(bindL·ys·g)"
  by (induct xs, simp_all add: appendL_appendL)

lemma bindL_bindL: "bindL·(bindL·xs·g)·h = bindL·xs·(Λ x. bindL·(g·x)·h)"
  by (induct xs, simp_all add: bindL_appendL)
```

7.2.3 Applicative functors and laws for zip

The functor and monad operations, `fmap`, `return` and `(>>=)`, are not the only functions on lazy lists with standard algebraic laws that we ought to verify. The *applicative functors* are another algebraic class of which lazy lists can be made an

```
class Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b   -- left-associative

             pure id <*> v = v                      -- Identity
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)        -- Composition
         pure g <*> pure x = pure (g x)             -- Homomorphism
              u <*> pure y = pure (\g -> g y) <*> u -- Interchange
```

Figure 7.5: Haskell class `Applicative`, with applicative functor laws

instance. The particular instance discussed here is based on the standard Haskell functions `repeat` and `zipWith`; in this section we prove that these operations satisfy the appropriate *applicative functor laws*.

The class of applicative functors for Haskell was introduced recently by McBride and Paterson [MP08]. An applicative functor is more than a functor, but less than a monad: Applicative functors support sequencing of effects, but not binding. The class is defined in Haskell as shown in Fig. 7.5. It fixes two functions: First, `pure` lifts an ordinary value into the applicative functor type; it denotes a computation with no effects, much like `return` for monads. Second, the left-associative infix operator `(<*>)` takes two computations, respectively yielding a function and an argument, and applies them together, sequencing their effects. All reasonable implementations of `pure` and `(<*>)` are expected to satisfy the four laws listed in Fig. 7.5.[1]

There is more than one way to instantiate class `Applicative` for the lazy list type. One possibility, which works for any monad, is to define `pure` and `(<*>)` in terms of the monadic operations `return` and `(>>=)`, as shown below.

[1] In the composition law, `(.) :: (b -> c) -> (a -> b) -> a -> c` denotes function composition.

```
repeat :: a -> [a]
repeat x = x : repeat x

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x : xs) (y : ys) = f x y : zipWith f xs ys
zipWith f _ _ = []

instance Applicative [] where
  pure x = repeat x
  fs <*> xs = zipWith id fs xs
```

Figure 7.6: Zip-style applicative functor instance for lazy lists

```
instance Applicative [] where
  pure x = return x
  fs <*> xs = fs >>= (\f -> xs >>= (\x -> return (f x)))
```

With these definitions, the four applicative functor laws can be proven by rewriting with the monad laws. An applicative functor instance can be derived from any monad in the same way. In terms of demonstrating proof techniques in HOLCF, this instantiation is rather unpromising.

For our purposes, a different instantiation using `repeat` and `zipWith` offers a more interesting verification challenge for HOLCF '11. The full code for the applicative functor instance is shown in Fig. 7.6. In this version, sequencing is done by taking a list of functions and a list of arguments, and applying them pointwise. The "effect" being sequenced here is essentially the dependence of values on their positions in the list. A pure computation (with no such "effect") is then represented as an infinite list whose elements are all the same.

7.2.4 Verifying the applicative functor laws

As with the functor and monad laws, the first step in verifying the applicative laws is to define the operations using Fixrec. We define the HOLCF functions repeatL and zipL following the Haskell definitions of repeat and zipWith.

```
fixrec repeatL :: "'a → 'a llist"
  where [simp del]: "repeatL·x = LCons·x·(repeatL·x)"

fixrec zipL :: "('a → 'b → 'c) → 'a llist → 'b llist → 'c llist"
  where "zipL·f·(LCons·x·xs)·(LCons·y·ys) = LCons·(f·x·y)·(zipL·f·xs·ys)"
  | (unchecked) "zipL·f·xs·ys = LNil"
```

There are a couple of annotations on these definitions that require explanation: First, because repeatL does not pattern match on a constructor, its defining equation would loop if used as a rewrite rule; for this reason, we declare it with [simp del] to remove it from the simplifier. In proofs, we must apply the rule repeatL.simps manually as needed. Second, the specification of zipL includes a catch-all case that returns LNil. This equation is not provable as a theorem, because it only applies when the other equation fails to match; thus we must declare it as (**unchecked**). The theorem list zipL.simps then only includes the first equation.

So far, Fixrec has only provided one rewrite rule about zipL. Doing proofs about zipL will require a few more, which we get using fixrec_simp.

```
lemma zipL_extra_simps [simp]:
  "zipL·f·⊥·ys = ⊥"
  "zipL·f·LNil·ys = LNil"
  "zipL·f·(LCons·x·xs)·⊥ = ⊥"
  "zipL·f·(LCons·x·xs)·LNil = LNil"
by fixrec_simp+
```

We define the HOLCF infix operator ⋄ to represent the Haskell operator (<*>) on lazy lists. Defining it as a syntactic abbreviation means that we can reason about it using the lemmas we already have for zipL.

```
abbreviation apL (infixl "⋄" 70)
  where "fs ⋄ xs ≡ zipL·ID·fs·xs"
```

With these definitions, we are now ready to consider the proofs of the applicative functor laws. We will start with the identity law, where we must prove that repeatL·ID ⋄ xs = xs. Induction on xs yields the following subgoals:

```
goal (4 subgoals):
 1. adm (λa. repeatL·ID ⋄ a = a)
 2. repeatL·ID ⋄ ⊥ = ⊥
 3. repeatL·ID ⋄ LNil = LNil
 4. ⋀a xs. repeatL·ID ⋄ xs = xs ⟹ repeatL·ID ⋄ LCons·a·xs = LCons·a·xs
```

The admissibility condition can be solved automatically by simp. But in the other goals, the rewrites we have for zipL do not apply yet. To use the rewrite rules we have for zipL, we must first manually unfold repeatL·ID one step, and then call simp. In this way we can solve all the remaining goals.

Alternatively, we can improve automation by defining some additional rewrite rules for zipL and repeatL. If one argument to zipL is repeatL, and the other is a constructor, we can unfold repeatL to make progress.

```
lemma zipL_repeatL_simps [simp]:
  "zipL·f·(repeatL·x)·⊥ = ⊥"
  "zipL·f·(repeatL·x)·LNil = LNil"
  "zipL·f·(repeatL·x)·(LCons·y·ys) = LCons·(f·x·y)·(zipL·f·(repeatL·x)·ys)"
  "zipL·f·(LCons·x·xs)·(repeatL·y) = LCons·(f·x·y)·(zipL·f·xs·(repeatL·y))"
by (subst repeatL.simps, simp)+
```

With these new rewrite rules in place, the proof of the identity law is now completely automatic, by induction followed by simplification.

```
lemma llist_identity: "repeatL·ID ⋄ xs = xs"
  by (induct xs, simp_all)
```

The zipL_repeatL_simps rules also make it possible to prove the interchange law with a similar level of automation:

```
lemma llist_interchange: "fs ⋄ repeatL·x = repeatL·(Λ f. f·x) ⋄ fs"
  by (induct fs, simp_all)
```

The composition law repeatL·cfcomp ⋄ fs ⋄ gs ⋄ xs = fs ⋄ (gs ⋄ xs) can also be proven by induction, although the proof is complicated by the fact that it mentions not one, but three list variables. In the proof script below, we induct over the first list, fs. The ⊥ and LNil cases can be solved automatically, but the LCons case requires extra case analyses on gs and xs. The arbitrary: gs xs option generalizes the inductive hypothesis by universally quantifying over the other two lists; this is because we will need the inductive hypothesis to apply not to gs and xs, but to the tails of those lists.

```
lemma llist_composition: "repeatL·cfcomp ◇ fs ◇ gs ◇ xs = fs ◇ (gs ◇ xs)"
  by (induct fs arbitrary: gs xs, simp_all,
      case_tac gs, simp_all, case_tac xs, simp_all)
```

7.2.5 Coinductive proof methods

Of all the four applicative functor laws for lazy lists, the most challenging to prove is the homomorphism law, repeatL·f ◇ repeatL·x = repeatL·(f·x). The technique of induction that we used for the other three laws will not work here—the homomorphism law does not mention any list variables to induct over! Instead, we must rely on *coinductive* proof methods: While inductive methods reason about the structure of *input* to functions, coinductive methods consider the structure of the *output*. Various such proof methods are surveyed by Gibbons and Hutton [GH05]; we will consider a few of those methods here, as applied to proving the homomorphism law in HOLCF '11.

First we consider a proof method that Gibbons and Hutton call the *approximation lemma* [HG01, GH05]; in HOLCF it is known as the *take lemma*. This method uses the function llist_take :: nat ⇒ 'a llist → 'a llist generated by the DOMAIN package, which satisfies the following specification:

```
theorem llist.take_rews:
  "llist_take 0 = ⊥"
  "llist_take (Suc n)·LNil = LNil"
  "llist_take (Suc n)·(LCons·x·xs) = LCons·x·(llist_take n·xs)"
```

The take lemma lets us prove that two lazy lists are equal, if we can show that llist_take n can never distinguish them for any n.

```
theorem llist.take_lemma: "(⋀n. llist_take n·x = llist_take n·y) ⟹ x = y"
```

In practice, an application of llist.take_lemma is usually followed by induction on n. If we take this approach to proving the homomorphism law, we are then left with the following two subgoals:

```
goal (2 subgoals):
 1. llist_take 0·(repeatL·f ◇ repeatL·x) = llist_take 0·(repeatL·(f·x))
 2. ⋀n. llist_take n·(repeatL·f ◇ repeatL·x) = llist_take n·(repeatL·(f·x)) ⟹
    llist_take (Suc n)·(repeatL·f ◇ repeatL·x) = llist_take (Suc n)·(repeatL·(f·x))
```

The first subgoal can be solved automatically, because both sides simplify to $\perp$. However, before the second subgoal can be simplified further, we must manually unfold by one step each of the three applications of repeatL. After these manual steps, the simplifier can finish the proof. All together, we get a proof script with about seven steps.

Perhaps a different proof method can yield better automation? The next method we will try is *fixed point induction*, which reasons about the structure of recursive calls of a function (repeatL in this case). As described in Chapter 3, a fixed point induction rule is generated for each function defined by Fixrec.

```
theorem repeatL.induct:
  "⟦adm P; P ⊥; ⋀r. P r ⟹ P (Λ x. LCons·x·(r·x))⟧ ⟹ P repeatL"
```

We can prove the homomorphism law repeatL·f ◇ repeatL·x = repeatL·(f·x) using repeatL.induct. A tempting idea is to try inducting simultaneously over all three occurrences of repeatL in parallel, but unfortunately this does not work: Each occurrence of repeatL has a different type, and the predicate P in repeatL.induct can only abstract over one of them at a time. Next we might try to induct over a single function in the original goal, but we hit another problem: For the base case, exactly one occurrence of repeatL will be replaced with $\bot$, leaving an unprovable goal.

We can still make progress if we apply an antisymmetry rule before trying fixed point induction. In each of the two subgoals, we can induct over an occurrence of repeatL on the left-hand side of the inequality, yielding provable base cases.

```
goal (2 subgoals):
 1. repeatL·f ◇ repeatL·x ⊑ repeatL·(f·x)
 2. repeatL·(f·x) ⊑ repeatL·f ◇ repeatL·x
```

Because we are only inducting over one occurrence of repeatL at a time, this means that in the inductive step, the other occurrences of repeatL will not be unfolded for us; we will have to do this manually. Ultimately we end up with a completed proof script that is even longer and more complicated than the take lemma proof.

Things would be much easier if we could do simultaneous fixed point induction! This would free us from having to do the antisymmetry step and the manual unfolding steps, yielding a much shorter, more automated proof. It turns out that there is a way to accomplish this, if we are willing to modify our function definitions a bit.

The idea is to augment repeatL with a new parameter that places a limit on the recursion depth. We can then redefine the original repeatL in terms of the depth-limited version by calling it with an infinite depth limit. To model the (possibly infinite) depth values, we define a domain depth with a single lazy constructor DSuc representing successor; $\bot$ represents zero. We use FIXREC to define unlimited as the fixed point of DSuc.

```
domain depth = DSuc (lazy depth)

fixrec unlimited :: "depth"
  where [simp del]: "unlimited = DSuc·unlimited"
```

Next we define repeatL_depth as a depth-limited version of repeatL. Recursive calls decrement the depth limit by one. We can use fixrec_simp to prove that repeatL_depth is strict in its first argument, which assures that the recursion stops when the depth limit reaches zero (i.e., $\bot$).

```
fixrec repeatL_depth :: "depth → 'a → 'a llist"
  where "repeatL_depth·(DSuc·n)·x = LCons·x·(repeatL_depth·n·x)"
```

We define repeatL' as the depth-unlimited version of repeatL_depth. Using the rewrites for unlimited and repeatL_depth, we can prove that repeatL' satisfies the same defining equations as the original repeatL.

```
definition repeatL' :: "'a → 'a llist"
  where "repeatL' = repeatL_depth·unlimited"
```

With these definitions in place, we want to show that repeatL' satisfies the homomorphism law, repeatL'·f ⋄ repeatL'·x = repeatL'·(f·x). We start by unfolding the definition of repeatL' to reveal applications of repeatL_depth to unlimited:

```
goal (1 subgoal):
 1. repeatL_depth·unlimited·f ⋄ repeatL_depth·unlimited·x
    = repeatL_depth·unlimited·(f·x)
```

Each occurrence of repeatL_depth in the goal has a different type, so fixed point induction on repeatL_depth would bring the same difficulties as before. But we have another option now: We can do fixed point induction on the depth parameter unlimited, using the Fixrec-provided rule unlimited.induct. Because all depth parameters have the same type, we can abstract over all three simultaneously.

```
theorem unlimited.induct:
  "⟦adm P; P ⊥; ⋀x. P x ⟹ P (DSuc·x)⟧ ⟹ P unlimited"
```

The rest of the proof is handled automatically by the simplifier. At last, we have achieved the high level of proof automation we were aiming for:

```
lemma llist_homomorphism: "repeatL'·f ⋄ repeatL'·x = repeatL'·(f·x)"
  unfolding repeatL'_def by (rule unlimited.induct, simp_all)
```

Fixed point induction with depth-limited functions looks like a promising general technique for automating HOLCF proofs, but its uses are still in the experimental stage. Eventually it might be beneficial to have the Fixrec package automatically generate depth-limited versions of all recursive functions, but this is left for future work.

7.3 A CONCURRENCY MONAD

The previous section gave an example of a simple datatype definition, with relatively easy proofs. In this section, we will discuss a significantly more complex monadic type, which will give the reasoning infrastructure of HOLCF '11 much more of a workout. In this way, we will demonstrate how well HOLCF '11 scales up to handle verification tasks that are beyond the scope of many other theorem-proving systems, including earlier versions of HOLCF.

The particular datatype considered in this section is a monad that is designed to model concurrent computations. It combines three different kinds of effects:

- Resumptions, to keep track of suspended threads of computations
- State, to allow multiple threads to communicate with each other
- Nondeterminism, to model computations with unpredictable evaluation order

The concurrency monad that we will verify here is identical to the one used by Papaspyrou [Pap01a] to model the semantics of a language with concurrency primitives. The monad uses powerdomains (see Chapter 5) to model nondeterminism, so it is not a direct translation from any monad definable in Haskell. However, it is still directly relevant for verification of Haskell programs, as a model of Haskell's primitive `ST` or `IO` monads [Thi95].

7.3.1 Composing monads

Many well-known monads encode a single, specific kind of effect—such as error handling, mutable state, string output, or nondeterminism. To model computations with one kind of effect, programmers can simply use one of these standard monads. But to model computations with a unique *combination* of effects, a programmer has two choices: Either write a new complex monad type all at once, by hand; or build the monad in a modular fashion, by combining a standard base monad with one or more standard *monad transformers* [Mog89, LHJ95]. A monad transformer is simply a monad that is parameterized over another, "inner" monad. The transformed monad supports all the effects of the inner monad, and adds more of its own. Multiple monad transformers can be layered to combine as many features as the programmer needs.

In this section we will follow Papaspyrou [Pap01a] in defining our concurrency monad using monad transformers. Specifically, we start with the convex powerdomain to model nondeterminism. Next we wrap this in a state monad transformer, and finally with a resumption monad transformer.

Using monad transformers not only makes it easier to write the monad operations, it also offers a nice way to structure the verification proofs. After introducing the state monad transformer (§7.3.2), we will then discuss the verification of a state/nondeterminism monad (§7.3.3). Then, after covering the resumption monad transformer (§7.3.4), we will see proofs for the full concurrency monad (§7.3.5–§7.3.7).

Besides the usual functor and monad operations, we will also define and verify an operation for nondeterministically interleaving two computations. Much like the `zipWith` operation on lazy lists, the interleaving operator can be proven to satisfy the applicative functor laws (§7.3.8).

```
newtype State s a = MkState { runState :: s -> (a, s) }

instance Functor (State s) where
  fmap f c = MkState
    (\s -> let (x, s’) = runState c s in (f x, s’))

instance Monad (State s) where
  return x = MkState (\s -> (x, s))
  c >>= k = MkState
    (\s -> let (x, s’) = runState c s in runState (k x) s’)
```

Figure 7.7: Haskell definition of state monad

7.3.2 State monad transformer

The *state monad* is used for computations that may imperatively read and write to a location in memory. The Haskell type `State s a` (Fig. 7.7) is represented as a function that takes an initial state of type `s`, and returns a pair containing a result of type `a` and a final state.

The *state monad transformer* replaces the function type `s -> (a, s)` with `s -> m (a, s)`, for some monad `m`. This allows the function on states to have some additional side-effects, depending on the choice of inner monad. The monad operations on `StateT s m` are defined similarly to those for `State s`, but they additionally include calls to the underlying monad operations on type `m` (Fig. 7.8).

Recall the `ChoiceMonad` type class, discussed previously in Chapter 5; it extends the `Monad` type class with an additional binary choice operator. We can define a binary choice operator on `StateT s m` in terms of the choice operator on monad `m`, as shown in Fig. 7.9.

All of these operations on type `StateT s m` should satisfy some equational laws, assuming that the operations on the inner monad `m` also satisfy the appropriate laws. Verifying these laws is the subject of the next section.

```
newtype StateT s m a = MkStateT { runStateT :: s -> m (a, s) }

instance (Functor m) => Functor (StateT s m) where
  fmap f c = MkStateT
    (\s -> fmap (\(x, s’) -> (f x, s’)) (runStateT c s))

instance (Monad m) => Monad (StateT s m) where
  return x = MkStateT (\s -> return (x, s))
  c >>= k = MkStateT
    (\s -> runStateT c s >>= \(x, s’) -> runStateT (k x) s’)
```

Figure 7.8: Haskell definition of state monad transformer

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

instance (ChoiceMonad m) => ChoiceMonad (StateT s m) where
  c1 |+| c2 = MkStateT (\s -> runStateT c1 s |+| runStateT c2 s)
```

Figure 7.9: Haskell `ChoiceMonad` class, with instance for state monad transformer

7.3.3 Verifying a state/nondeterminism monad

In this section, we will verify an instance of the state monad transformer, using the convex powerdomain as the inner monad. This combination yields a monad that implements two out of the three effects that we ultimately want: state and nondeterminism. Building on the existing powerdomain operations, we define the monad operations on this new type, along with a nondeterministic choice operator. We also use the powerdomain laws to help derive the usual laws about the new operations. (Refer to Chapter 5 for the operations on the convex powerdomain, and the laws that they satisfy.)

In HOLCF, we define a type ('s, 'a) N to model the state monad transformer applied to the convex powerdomain, with state type 's and result type 'a. To avoid having to deal with HOLCF equivalents of the `MkStateT` and `runStateT` functions, we define N as a type synonym. The Haskell pair type `(a, s)` (which is lazy in both arguments) is modeled as a HOLCF strict product of two lifted types. We then define the HOLCF functions mapN, unitN, bindN, and plusN to model the Haskell functions `fmap`, `return`, `(>>=)`, and `(|+|)`, respectively. Their definitions are shown in Fig. 7.10.

Figure 7.11 contains a list of all the relevant theorems about these operations on type ('s, 'a) N that were proved in this case study. Included are the laws for functors, monads, and choice monads—in fact, all of the powerdomain laws from Chapter 5 are satisfied by the N monad. Because ('s, 'a) N is not a recursive type, none of the proofs require any form of induction, merely case analysis. Most of them derive fairly directly from the corresponding properties of the underlying powerdomain type. All the proofs are straightforward (many are one line), and so we omit the details.

```
type_synonym ('s, 'a) N = "'s → ('a⊥ ⊗ 's⊥)♮"

definition mapN :: "('a → 'b) → ('s, 'a) N → ('s, 'b) N"
  where "mapN = (Λ f. cfun_map·ID·(convex_map·(sprod_map·(u_map·f)·ID)))"

definition unitN :: "'a → ('s, 'a) N"
  where "unitN = (Λ x. (Λ s. convex_unit·(:up·x, up·s:)))"

definition bindN :: "('s, 'a) N → ('a → ('s, 'b) N) → ('s, 'b) N"
  where "bindN = (Λ c k. (Λ s. convex_bind·(c·s)·(Λ (:up·x, up·s':). k·x·s')))"

definition plusN :: "('s, 'a) N → ('s, 'a) N → ('s, 'a) N"
  where "plusN = (Λ a b. (Λ s. convex_plus·(a·s)·(b·s)))"
```

Figure 7.10: Functor, monad, and choice operations on state/nondeterminism monad

7.3.4 Resumption monad transformer

The *resumption monad transformer* [Pap01a] augments an inner monad with the ability to suspend, resume, and interleave threads of computations. In Haskell, we define the type `ResT m a` to model resumptions with inner monad `m` and result type `a`.

```
data ResT m a = Done a | More (m (ResT m a))
```

The value `Done x` represents a computation that has run to completion, yielding the result `x`. The value `More c` represents a suspended computation that still has more work to do: When `c` is evaluated, it may produce some side-effects (according to the monad `m`) and eventually yields a new resumption of type `ResT m a`. A good way to think about resumptions is as threads in a cooperative multitasking system: A running thread may either terminate (`Done x`) or voluntarily yield to the operating system, waiting to be resumed later (`More c`).

```
lemma mapN_ID:
  "mapN·ID = ID"

lemma mapN_mapN:
  "mapN·f·(mapN·g·c) = mapN·(Λ x. f·(g·x))·c"

lemma bindN_unitN:
  "bindN·(unitN·x)·f = f·x"

lemma mapN_conv_bindN:
  "mapN·f·c = bindN·c·(unitN oo f)"

lemma bindN_unitN_right:
  "bindN·c·unitN = c"

lemma bindN_bindN:
  "bindN·(bindN·c·f)·g = bindN·c·(Λ x. bindN·(f·x)·g)"

lemma mapN_plusN:
  "mapN·f·(plusN·a·b) = plusN·(mapN·f·a)·(mapN·f·b)"

lemma plusN_commute:
  "plusN·a·b = plusN·b·a"

lemma plusN_assoc:
  "plusN·(plusN·a·b)·c = plusN·a·(plusN·b·c)"

lemma plusN_absorb:
  "plusN·a·a = a"
```

Figure 7.11: Laws satisfied by operations on state/nondeterminism monad

```
data ResT m a = Done a | More (m (ResT m a))

instance (Functor m) => Functor (ResT m) where
  fmap f (Done x) = Done (f x)
  fmap f (More c) = More (fmap (fmap f) c)

instance (Functor m) => Monad (ResT m s) where
  return x     = Done x
  Done x >>= k = k x
  More c >>= k = More (fmap (\r -> r >>= k) c)
```

Figure 7.12: Haskell definition of resumption monad transformer

The code for the functor and monad instances is given in Fig. 7.12. Note that the `fmap` and `(>>=)` are defined by using the `fmap` from the underlying type constructor `m` on the recursive calls.

The resumption monad transformer can be used to nondeterministically interleave two computations, when used with an inner monad that provides a binary choice operator. We can define a Haskell function `zipR` that randomly interleaves two computations, and combines their results (Fig. 7.13). The idea is that as long as at least one of the two computations has the form `More c`, `zipR` chooses one, runs it for one step, and repeats. When both computations have the form `Done x`, it combines the results using function `f`.

The name of the function `zipR` has been chosen to be reminiscent of the standard Haskell list function `zipWith`, which was covered earlier in this chapter. The two functions have similar types, and it turns out that they also satisfy many of the same equational laws: Like `zipWith`, `zipR` forms the basis of an applicative functor instance. After formalizing and verifying the functor and monad operations in the next few sections, we will return to a verification of `zipR` in Sec. 7.3.8.

```
zipR :: (ChoiceMonad m) =>
    (a -> b -> c) -> ResT m a -> ResT m b -> ResT m c
zipR f (Done x1) (Done x2) = Done (f x1 x2)
zipR f (Done x1) (More c2) = More
                             (fmap (\r -> zipR f (Done x1) r) c2)
zipR f (More c1) (Done x2) = More
                             (fmap (\r -> zipR f r (Done x2) r) c1)
zipR f (More c1) (More c2) = More
                             (fmap (\r -> zipR f (More c1) r) c2 |+|
                              fmap (\r -> zipR f r (More c2)) c1)
```

Figure 7.13: Haskell definition of nondeterministic interleaving operator

7.3.5 Defining the full concurrency monad

Having already formalized the N monad for state and nondeterminism, only one step remains: We must define the final monad by combining the N monad with a resumption monad transformer. The domain definition shown below is handled easily by the Domain package:

domain ('s, 'a) R = Done (**lazy** "'a") | More (**lazy** "('s, ('s, 'a) R) N")

This type definition exercises some unique abilities of the HOLCF '11 Domain package. First of all, note that the definition uses indirect recursion: The recursive occurrence of ('s, 'a) R is not actually an argument type of a constructor, but is wrapped inside the N monad type—which is itself a combination of lifting, strict product, convex powerdomain, and the continuous function space. For comparison, the original HOLCF '99 Domain package was not designed to handle indirect recursion at all [Ohe97]. The Isabelle/HOL Datatype package can handle *some* indirect-recursive definitions by transforming them (internally) into equivalent mutually recursive definitions, but this only works for indirect recursion with other HOL datatypes. If the Domain package had been implemented with the same technique, it would not have been able to define domain R—its indirect recursion cannot be translated away as mutual recursion, because it involves a powerdomain type. To the best of the author's knowledge, the HOLCF '11 DOMAIN package is the first formal reasoning system that can handle such a type definition.

Due to the indirect recursion, however, the package warns us that it has not attempted to generate a high-level induction rule (i.e., one stated in terms of the constructors Done and More). The only induction rule generated for indirect-recursive domains is a low-level one in terms of take functions.

```
theorem R.take_induct: "⟦adm P; ⋀n. P (R_take n·x)⟧ ⟹ P x"
```

We can use the low-level take induction rule to generate our own high-level induction rule. Depending on the particular domain definition, there may be more than one sensible formulation of a high-level induction rule for a given indirect-recursive datatype. Generating one that will work well for the proofs about R is the subject of the next section.

7.3.6 Induction rules for indirect-recursive domains

There are several different ways to express induction principles over indirect-recursive datatypes. The goal of this section is to find a style of induction rule that will work well for proving results about the domain R. To that end, we will consider a few alternatives, using a relatively simple Isabelle/HOL datatype of trees as a basis for examples.

```
datatype 'a tree = Leaf "'a" | Branch "'a tree list"
```

For indirect-recursive datatypes like 'a tree, the DATATYPE package generates an induction rule similar to the ones produced for mutually recursive datatype definitions. Types 'a tree and 'a tree list are treated as two mutually defined datatypes; the induction rule has one predicate for each of them.

```
theorem tree.induct:
  fixes P :: "'a tree ⇒ bool" and Q :: "'a tree list ⇒ bool"
  assumes "⋀x. P (Leaf x)"
  assumes "⋀ts. Q ts ⟹ P (Branch ts)"
  assumes "Q []"
  assumes "⋀t ts. ⟦P t; Q ts⟧ ⟹ Q (t # ts)"
  shows "P t ∧ Q ts"
```

This form of induction rule makes sense for reasoning about pairs of mutually defined functions, such as these map functions for trees and lists of trees:

```
primrec tree_map :: "('a ⇒ 'b) ⇒ 'a tree ⇒ 'b tree"
  where "tree_map f (Leaf x) = Leaf (f x)"
  | "tree_map f (Branch ts) = Branch (tree_list_map f ts)"
and tree_list_map :: "('a ⇒ 'b) ⇒ 'a tree list ⇒ 'b tree list"
  where "tree_list_map f [] = []"
  | "tree_list_map f (t # ts) = tree_map f t # tree_list_map f ts"
```

The mutually recursive induction rule tree.induct is a good match for proving properties about mutually recursive functions like tree_map and tree_list_map, because we can have a predicate P mentioning tree_map and a predicate Q mentioning tree_list_map.

On the other hand, suppose we have a single recursive function defined like this, where the newly-defined function is mapped over a list in the recursive case:

```
fun tree_map' :: "('a ⇒ 'b) ⇒ 'a tree ⇒ 'b tree"
  where "tree_map' f (Leaf x) = Leaf (f x)"
  | "tree_map' f (Branch ts) = Branch (map (tree_map' f) ts)"
```

With this style of definition, the mutual-recursion-style induction rule is awkward to use. An alternative form with a single predicate would be preferable, such as the rule tree_all_induct shown below.

```
lemma tree_all_induct:
  assumes "⋀x. P (Leaf x)"
  assumes "⋀ts. list_all P ts ⟹ P (Branch ts)"
  shows "P t"
```

Here the function list_all :: ('a ⇒ bool) ⇒ 'a list ⇒ bool is a predicate former from the list library; list_all P ts asserts that predicate P holds for all elements of the list ts. We will refer to this style of rule, which refers to an *all* predicate on some datatype, by the name *all*-induction.

As stated earlier, the recursive domain ('s, 'a) R is not equivalent to any mutually inductive domain definition, so we cannot hope to produce a mutual induction rule for type ('s, 'a) R in the style of tree.induct. There is more hope for an all-induction rule, because it may be possible to generalize predicate formers like list_all to other type constructors (like powerdomains) that are not necessarily datatypes.

However, it turns out that there is yet another form of induction rule that generalizes even better: Instead of requiring a predicate former like list_all, we can express an induction rule that needs nothing more than a map function. We will refer to such rules as *map*-induction rules. A map-induction rule for the 'a tree datatype is shown below.

```
lemma tree_map_induct:
  fixes P :: "'a tree ⇒ bool"
  assumes 1: "⋀x. P (Leaf x)"
  assumes 2: "⋀f ts. (∀t::'a tree. P (f t)) ⟹ P (Branch (map f ts))"
  shows "P t"
```

Map-induction rules in the style of tree_map_induct generalize readily to most any indirect-recursive domain definition. Recall from Chapter 4 that map functions are already required in order to define indirect-recursive domains. Map-induction rules also work well in practice for proving properties of recursively-defined functions, as we will see later on.

Map-induction rules for indirect-recursive domains can be derived from the low-level take-induction rules in a straightforward way. We will now step through the derivation of the map-induction rule for domain ('s, 'a) R, which is shown below.

```
lemma R_induct:
  fixes P :: "('s, 'a) R ⇒ bool"
```

```
assumes adm: "adm P"
assumes bottom: "P ⊥"
assumes Done: "⋀x. P (Done·x)"
assumes More: "⋀p c. (⋀r::('s, 'a) R. P (p·r)) ⟹ P (More·(mapN·p·c))"
shows "P r"
```

The proof starts by applying the low-level take induction rule R.take_induct: Because P is admissible, to prove P r it is sufficient to show that P (R_take n·r) for all natural numbers n. The remainder of the proof proceeds by showing ∀r. P (R_take n·r) by induction on n.

In the base case n = 0, we have R_take n·r = ⊥. The goal ∀r. P ⊥ can then be solved immediately using the assumptions.

In the inductive case n = Suc n', we proceed by case analysis on r. The three possibilities are r = ⊥, r = Done·x, and r = More·c. By the definition of R_take, we then have R_take n·r = ⊥, Done·x, or More·(mapN·(R_take n')·c), respectively. Each of these subcases can be discharged using the assumptions together with the inductive hypothesis.

7.3.7 Verifying functor and monad laws

Recall the Haskell code for the resumption monad transformer from Fig. 7.12. We translate these definitions directly into HOLCF using the Fixrec package, as shown below. Note that we do not define a separate returnR function in HOLCF; the relevant properties are stated directly in terms of the Done constructor.

```
fixrec mapR :: "('a → 'b) → ('s, 'a) R → ('s, 'b) R"
  where mapR_Done: "mapR·f·(Done·x) = Done·(f·x)"
  | mapR_More: "mapR·f·(More·n) = More·(mapN·(mapR·f)·n)"

fixrec bindR :: "('s, 'a) R → ('a → ('s, 'b) R) → ('s, 'b) R"
  where bindR_Done: "bindR·(Done·x)·k = k·x"
  | bindR_More: "bindR·(More·c)·k = More·(mapN·(Λ r. bindR·r·k)·c)"
```

In addition to the defining equations, we also need strictness rules for mapR and bindR; these are provided by fixrec_simp.

```
lemma mapR_strict [simp]: "mapR·f·⊥ = ⊥"
  by fixrec_simp

lemma bindR_strict [simp]: "bindR·⊥·k = ⊥"
  by fixrec_simp
```

Now we will see how well our induction rule R_induct from the previous section works in practice, by using it to make highly automated proofs of the functor and monad laws. We examine one proof in detail, showing that mapR preserves function composition.

```
lemma mapR_mapR: "mapR·f·(mapR·g·r) = mapR·(Λ x. f·(g·x))·r"
  apply (induct r rule: R_induct)
```

Applying the induction rule leaves us with four subgoals: an admissibility side condition, base cases for ⊥ and Done·x, and an inductive case for More·(mapN·p·c).

```
goal (4 subgoals):
 1. adm (λa. mapR·f·(mapR·g·a) = mapR·(Λ x. f·(g·x))·a)
 2. mapR·f·(mapR·g·⊥) = mapR·(Λ x. f·(g·x))·⊥
 3. ⋀x. mapR·f·(mapR·g·(Done·x)) = mapR·(Λ x. f·(g·x))·(Done·x)
 4. ⋀p c. (⋀r. mapR·f·(mapR·g·(p·r)) = mapR·(Λ x. f·(g·x))·(p·r)) ⟹
  mapR·f·(mapR·g·(More·(mapN·p·c))) = mapR·(Λ x. f·(g·x))·(More·(mapN·p·c))
```

The first three subgoals can be solved directly by the simplifier, using the defining properties of mapR. If we simplify the conclusion of the final subgoal using mapR_More, it reduces to the following:

```
mapN·(mapR·f)·(mapN·(mapR·g)·(mapN·p·c)) =
  mapN·(mapR·(Λ x. f·(g·x)))·(mapN·p·c)
```

We can see that the subgoal now contains instances of mapN applied to mapN. If we rewrite using the rule mapN_mapN, the subgoal reduces further:

```
mapN·(Λ x. mapR·f·(mapR·g·(p·x)))·c =
  mapN·(Λ x. mapR·(Λ x. f·(g·x))·(p·x))·c
```

```
lemma mapR_mapR: "mapR·f·(mapR·g·r) = mapR·(Λ x. f·(g·x))·r"
  by (induct r, simp_all add: mapN_mapN)

lemma mapR_ID: "mapR·ID·r = r"
  by (induct r, simp_all add: mapN_mapN eta_cfun)

lemma bindR_Done_right: "bindR·r·Done = r"
  by (induct r, simp_all add: mapN_mapN eta_cfun)

lemma mapR_conv_bindR: "mapR·f·r = bindR·r·(Λ x. Done·(f·x))"
  by (induct r, simp_all add: mapN_mapN)

lemma bindR_bindR: "bindR·(bindR·r·f)·g = bindR·r·(Λ x. bindR·(f·x)·g)"
  by (induct r, simp_all add: mapN_mapN)
```

Figure 7.14: Functor and monad laws for concurrency monad

Finally, the remaining subgoal can now be solved using the inductive hypothesis. In the final proof script, we can perform all of these rewriting steps at once with a single call to the simplifier, yielding an easy one-line proof:

```
lemma mapR_mapR: "mapR·f·(mapR·g·r) = mapR·(Λ x. f·(g·x))·r"
  by (induct r rule: R_induct, simp_all add: mapN_mapN)
```

All of the functor and monad laws for the ('s, 'a) R monad shown in Fig. 7.14 have similar one-line proofs, using induction followed by simplification. The fact that this level of automation is possible demonstrates the general utility of map-induction.

7.3.8 Verification of nondeterministic interleaving

In this section we will see how to formalize and verify the nondeterministic interleaving operator in HOLCF '11. We model the Haskell function `zipR`, given previously in Fig. 7.13, as the HOLCF function zipR. The definition, using Fixrec, is shown in Fig. 7.15. The translation is mostly direct, but note that we fix the inner

```
fixrec zipR :: "('a → 'b → 'c) → ('s, 'a) R → ('s, 'b) R → ('s, 'c) R"
  where zipR_Done_Done:
    "zipR·f·(Done·x)·(Done·y) = Done·(f·x·y)"
  | zipR_Done_More:
    "zipR·f·(Done·x)·(More·b) = More·(mapN·(Λ r. zipR·f·(Done·x)·r)·b)"
  | zipR_More_Done:
    "zipR·f·(More·a)·(Done·y) = More·(mapN·(Λ r. zipR·f·r·(Done·y))·a)"
  | zipR_More_More:
    "zipR·f·(More·a)·(More·b) = More·
      (plusN·(mapN·(Λ r. zipR·f·(More·a)·r)·b)·(mapN·(Λ r. zipR·f·r·(More·b))·a))"
```

Figure 7.15: HOLCF definition of nondeterministic interleaving operator

monad `m` to be the N monad in HOLCF, so `fmap` and `(|+|)` translate to mapN and plusN, respectively.

In addition to the defining equations supplied by Fixrec, we can also prove lemmas stating that zipR is strict in its second and third arguments, with the aid of the fixrec_simp method.

In order to state the applicative functor laws, we need HOLCF equivalents for the Haskell operations `(<*>)` and `pure`. Just as we did for lazy lists, we define an abbreviation for zipR·ID with infix syntax to represent `(<*>)`. The constructor function Done takes the place of `pure`.

```
abbreviation apR (infixl "◇" 70)
  where "a ◇ b ≡ zipR·ID·a·b"
```

With the definitions in place, we can now start to prove the applicative functor laws. Surprisingly, the homomorphism law—which was by far the trickiest to verify for lazy lists—is the easiest to prove for the R monad. The law follows directly from zipR_Done_Done, one of the defining equations of zipR.

```
lemma R_homomorphism: "Done·f ◇ Done·x = Done·(f·x)"
  by simp
```

The proofs of the identity and interchange laws are also easy. They both have proofs similar to those for the functor and monad laws, by induction and simplification with mapN_mapN (and also eta-contraction, as needed).

```
lemma R_identity: "Done·ID ◇ r = r"
  by (induct r, simp_all add: mapN_mapN eta_cfun)
```

```
lemma R_interchange: "r ◇ Done·x = Done·(Λ f. f·x) ◇ r"
  by (induct r, simp_all add: mapN_mapN)
```

Of the four applicative functor laws, the associativity law stands out as the tricky one to prove. The fact that the law mentions three variables of type ('s, 'a) R necessarily complicates the proof.

```
lemma R_associativity: "Done·cfcomp ◇ r1 ◇ r2 ◇ r3 = r1 ◇ (r2 ◇ r3)"
```

For lazy lists, the proof of the associativity law only requires induction over one list variable, with case analysis on the other two. The reason this works is that in the definition of zipL, every argument decreases with every recursive call—specifically, zipL·(LCons·x·xs)·(LCons·y·ys) only makes a recursive call to zipL·xs·ys. So with a single induction over either xs or ys, we can be sure that the inductive hypothesis will apply to the result of any recursive call.

In contrast, recursive calls to zipR only decrease one argument, while the other stays the same—and it is not always the same one! Thus a single induction will not suffice. To prove the associativity law for zipR, we must perform nested inductions: First, induct over r1; within each case of the induction, proceed by another induction over r2; finally, prove each of *those* cases by induction over r3.

Without knowing beforehand whether it would work, a proof of R_associativity was attempted using this strategy: Do nested inductions as needed with R_induct, and discharge subgoals by simplification. The proof attempt was a real test of the map-induction principle of R_induct: An induction principle that is too weak could have easily led to a proof state with unprovable subgoals. Fortunately, this did not happen with the proof of R_associativity, and the proof was completed as planned. Although the proof is rather lengthy, it goes through without getting stuck, and without requiring additional lemmas.

Figure 7.16 gives a cleaned-up version of the complete proof script. To keep track of the numerous inductions and subcases, the proof is presented in Isabelle's structured-proof style. The command **proof** (induct ...) starts a new proof by induction. Within an induction proof, **case** (...) selects a subgoal to work on, assuming any inductive hypotheses and binding the new subgoal to the variable ?case. The **qed** command closes a block opened by **proof**; the variant **qed** simp_all says to prove any remaining subcases using the simplifier.

Recall that an induction with R_induct actually yields four subgoals: An admissibility condition, and cases for $\bot$, Done·x, and More·(mapN·p·c). If all three cases always required nested inductions to prove, this would have resulted in a proof script with $3^3 = 27$ separate subcases to discharge, in addition to the admissibility checks. Fortunately, there are some shortcuts—for example, because zipR is strict, any $\bot$ cases can be proved immediately without needing an inner induction.

The remaining cases are solvable with varying levels of effort. The Done/Done/Done case can be solved automatically by simplification, no extra lemmas needed. Subgoals where exactly two of the three computations are Done can be solved by simplifying with the additional rule mapN_mapN, as in the proofs of the functor and monad laws.

Cases where only one of the three computations is Done require a little more work. For example, in the Done/More/More case, we have the following inductive hypotheses and proof obligation:

$$\begin{aligned}
&\textstyle\bigwedge \mathsf{r3}.\ \mathsf{Done{\cdot}cfcomp} \diamond \mathsf{Done{\cdot}x1} \diamond \mathsf{More{\cdot}(mapN{\cdot}p2{\cdot}c2)} \diamond \mathsf{p3{\cdot}r3} = \\
&\quad \mathsf{Done{\cdot}x1} \diamond (\mathsf{More{\cdot}(mapN{\cdot}p2{\cdot}c2)} \diamond \mathsf{p3{\cdot}r3}) \\
&\textstyle\bigwedge \mathsf{r2\ r3}.\ \mathsf{Done{\cdot}cfcomp} \diamond \mathsf{Done{\cdot}x1} \diamond \mathsf{p2{\cdot}r2} \diamond \mathsf{r3} = \mathsf{Done{\cdot}x1} \diamond (\mathsf{p2{\cdot}r2} \diamond \mathsf{r3})
\end{aligned}$$

```
lemma R_associativity: "Done·cfcomp ◇ r1 ◇ r2 ◇ r3 = r1 ◇ (r2 ◇ r3)"
proof (induct r1 arbitrary: r2 r3)
  case (Done x1) thus ?case
  proof (induct r2 arbitrary: r3)
    case (Done x2) thus ?case
    proof (induct r3)
      case (More p3 c3) thus ?case (∗ Done/Done/More ∗)
        by (simp add: mapN_mapN)
    qed simp_all
  next
    case (More p2 c2) thus ?case
    proof (induct r3)
      case (Done x2) thus ?case (∗ Done/More/Done ∗)
        by (simp add: mapN_mapN)
    next
      case (More p3 c3) thus ?case (∗ Done/More/More ∗)
        by (simp add: mapN_mapN mapN_plusN)
    qed simp_all
  qed simp_all
next
  case (More p1 c1) thus ?case
  proof (induct r2 arbitrary: r3)
    case (Done x2) thus ?case
    proof (induct r3)
      case (Done x3) thus ?case (∗ More/Done/Done ∗)
        by (simp add: mapN_mapN)
    next
      case (More p3 c3) thus ?case (∗ More/Done/More ∗)
        by (simp add: mapN_mapN)
    qed simp_all
  next
    case (More p2 c2) thus ?case
    proof (induct r3)
      case (Done x3) thus ?case (∗ More/More/Done ∗)
        by (simp add: mapN_mapN mapN_plusN)
    next
      case (More p3 c3) thus ?case (∗ More/More/More ∗)
        by (simp add: mapN_mapN mapN_plusN plusN_assoc)
    qed simp_all
  qed simp_all
qed simp_all
```

Figure 7.16: Full proof of associativity for nondeterministic interleaving operator

```
goal (1 subgoal):
 1. Done·cfcomp ◇ Done·x1 ◇ More·(mapN·p2·c2) ◇ More·(mapN·p3·c3) =
    Done·x1 ◇ (More·(mapN·p2·c2) ◇ More·(mapN·p3·c3))
```

After inserting the inductive hypotheses and applying (simp add: mapN_mapN), we get stuck with the following unsolved goal:

```
goal (1 subgoal):
 1. ⟦⋀r3. More·(mapN·(Λ x. Done·(cfcomp·x1) ◇ p2·x)·c2) ◇ p3·r3 =
          Done·x1 ◇ (More·(mapN·p2·c2) ◇ p3·r3);
     ⋀r2 r3. Done·(cfcomp·x1) ◇ p2·r2 ◇ r3 = Done·x1 ◇ (p2·r2 ◇ r3)⟧
    ⟹ plusN·(mapN·(Λ r3. Done·x1 ◇ (More·(mapN·p2·c2) ◇ p3·r3))·c3)·
       (mapN·(Λ x. Done·x1 ◇ (p2·x ◇ More·(mapN·p3·c3)))·c2) =
       mapN·(Λ r. Done·x1 ◇ r)·
       (plusN·(mapN·(Λ r3. More·(mapN·p2·c2) ◇ p3·r3)·c3)·
        (mapN·(Λ r2. p2·r2 ◇ More·(mapN·p3·c3))·c2))
```

Note that the right-hand side of the conclusion has the form mapN·f·(plusN·x·y). Fortunately we have a rewrite rule mapN_plusN (Fig. 7.11) that matches this pattern. If we back up and try (simp add: mapN_mapN mapN_plusN), then the subgoal is discharged in one step. The general lesson about map-induction is this: Successful proofs require the map function to distribute over any other functions (such as mapN or plusN) that may surround recursive calls.

The More/More/More case also requires simplification with extra rewrite rules. We might start by trying (simp add: mapN_mapN mapN_plusN) again, which we used to solve the Done/More/More case. This leaves an unsolved goal, which is an equality between two very large terms—it is not worth the space to repeat it here. The one important detail about the leftover goal is that it has the form plusN·x·(plusN·y·z) = plusN·(plusN·x·y)·z, which is an instance of rule plusN_assoc (Fig. 7.11). By also including this rule in the call to the simplifier, the subgoal can be discharged in one step. Following this pattern, every subgoal is solved by simplification with an appropriate set of rewrite rules; this concludes the proof.

The proof of R_associativity using the map-induction rule R_induct shows that map-induction is not only useful for small or obvious lemmas. It is a strong enough reasoning principle to be useful for exploratory proving of complex theorems.

7.4 SUMMARY

Based on the case studies presented in this chapter, we can make some conclusions about the expressiveness and automation of HOLCF '11. In terms of expressiveness, we have seen that the definition packages of HOLCF '11 let users express a wide variety of programs in a direct and concise way. The Fixrec package provides input notation that is similar to Haskell syntax, making it easy to translate programs into HOLCF. Users can define arbitrary recursive functions with pattern matching, without having to use explicit fixed point combinators and without having to prove termination. Likewise, the new Domain package makes it easy to translate a wide variety of Haskell datatypes into HOLCF '11. In particular, the ability to define and reason about indirect-recursive datatypes opens up HOLCF to an important new class of programs, including many interesting Haskell monads. New theory libraries in HOLCF '11 also contribute to its expressiveness. Specifically, the powerdomain library expands HOLCF's range to include nondeterministic and concurrent programs.

In addition to expressiveness, HOLCF '11 provides a high level of proof automation. In the introduction we claimed that easy proofs should be almost completely automatic, and we have seen that many lemmas can be proved with one-line proof scripts, using induction and simplification. Instead of dealing with mundane details, proof effort can be focused on more interesting tasks, like identifying important lemmas. We also claimed that the automation helps to make larger, more complex proofs more feasible, by letting users focus on only the interesting parts of proofs. The proof of associativity for the nondeterministic interleaving operator is evidence of this: We can write a proof script where only those subcases with nontrivial proofs are considered explicitly; admissibility checks and trivial induction cases can be handled automatically, without having to mention them in the proof script.

7.5 COMPARISON TO RELATED WORK

We claim that HOLCF ’11 has an unprecedented combination of expressiveness, automation, and confidence. To substantiate this claim, we evaluate several other reasoning tools along these dimensions.

Informal verification. To start with, it may be instructive to compare the HOLCF ’11 concurrency monad case study from this chapter with manual proofs of the same theorems. As a companion to his conference paper [Pap01a], Papaspyrou provides detailed hand-written proofs of several properties about the resumption monad transformer in a technical report [Pap01b]. The proofs are based on a domain-theoretic model of recursive types, taking advantage of the bifinite structure of the resumption monad: Instead of reasoning about the recursively-defined domain directly, Papaspyrou instead works with the sequence of finite domains that approximate it. Results are then transferred from the finite domains to the full recursive domain in a separate step.

Informal reasoning provides no automation, but it gives a lot of flexibility in terms of expressiveness. With no tool-imposed limits, it is possible to reason about any kind of program that has a known mathematical or domain-theoretical model. However, informal proofs are limited in terms of confidence, because each proof must be read and understood in order to be trusted. This can become a major limitation as proofs grow in size.

Papaspyrou’s manual proofs of the functor and monad laws for the resumption monad, including auxiliary definitions and lemmas, take up about $9\frac{1}{2}$ pages of the document [Pap01b]. Papaspyrou did not attempt any manual proofs about the more complicated interleaving operation, but we can estimate the relative proof complexity by looking at the HOLCF scripts: Most of the functor and monad law proofs, whose informal proofs take almost a page each, have one- or two-line proofs in HOLCF. In comparison, the associativity law R_associativity has a 40-line HOLCF proof script (Fig. 7.16); at this ratio, one can imagine the amount of hand-written proof text it would take to prove R_associativity with a similar level of rigor. At this scale, it becomes difficult for a reader to check a written proof for correctness, which limits the confidence in manually-proven results.

Simpler domain-theoretic models can yield shorter informal proofs that are easier to understand and check. For example, the induction principles used in Bird's Haskell textbook [Bir98] and the approximation lemma of Hutton and Gibbons [HG01] are proof techniques based on a simpler category of cpos, and they yield simpler proofs. But of course, there is a tradeoff here with expressiveness, because simpler models can represent fewer datatypes and programs.

Formalizations of domain theory. The previous systems most similar to HOLCF '11 were of course earlier versions of HOLCF; we have already made numerous comparisons in the previous chapters. We have also discussed some features of Agerholm's similar HOL-CPO system in Chapters 2 and 6 [Age94]. The HOL-CPO system was comparable to HOLCF '95 in terms of its expressive power and proof automation. It also included a tree-based universal domain that was intended for constructing polynomial (i.e., sum-of-products) datatypes like lazy lists, although the process was not automated. The HOLCF '99 Domain package brought much more automation for defining datatypes; however, this was achieved at the expense of confidence, because of its axiomatic implementation. While HOL-CPO had very little automation for defining datatypes, it did at least preserve confidence by using a definitional approach.

More recently, Benton, et al. have formalized a significant amount of domain theory in the Coq theorem prover [BKV09]. They implemented sufficient machinery to construct solutions to recursive domain equations. The functionality is comparable to that provided by the universal domain and algebraic deflation libraries of HOLCF '11, although Benton, et al. use different methods to achieve the same result. Like HOLCF '11, their formalization is completely definitional, asserting no new axioms. However, their system provides little automation for verifying individual functional programs. Their main emphasis has been on doing programming-language meta-theory—i.e., building denotational models of other programming languages and reasoning about the models—rather than on verification of specific programs. To illustrate this point, note that their system lacks lambda-binder syntax (Λx. t) for writing continuous functions; instead users must compose functions from basic combinators like S and K.

General-purpose interactive theorem provers. In the earlier chapters, we have already discussed the relationship between HOLCF and the original LCF series of theorem provers. Having a first-order logic, the LCF provers had a less expressive property language than HOLCF. The definition packages in HOLCF '11 also yield improvements in expressiveness and automation over LCF, which did not provide packages for either recursive datatypes or functions.

On the other hand, more recent theorem provers in the HOL family (Isabelle/HOL, Gordon HOL/HOL4, HOL Light) do provide packages for defining datatypes and recursive functions [Mel89, Sli96, BW99]. Each supports an input syntax similar to most functional programming languages, and can be used to define many datatypes and functions that are commonly used by functional programmers. However, these packages generally provide only inductive datatypes, and allow only terminating functions (usually with either primitive or well-founded recursion). That is, unlike Haskell, datatypes never include any partial values (like ⊥) or infinite values. We can use higher order logic to reason about Haskell programs, but the reasoning is only valid if we restrict our consideration to the total, terminating fragment of the language [DGHJ06]. For example, $\forall \mathsf{xs}.\ \mathsf{reverse}\ (\mathsf{reverse}\ \mathsf{xs}) = \mathsf{xs}$ is a theorem in Isabelle/HOL, and although it does not hold for all lazy lists in Haskell, it is valid if we interpret the quantification as ranging over only finite, total lists. Haskell programs that produce or consume infinite or partial values are beyond the scope of such tools.

In addition to inductive datatypes, a few theorem provers also provide support for coinductive datatypes (or "codatatypes"), which include both finite and infinite values. The primary example is Coq, which has mature support for coinductive types [Gim96]. This feature lets users define programs that produce infinite values, as long as they are provably productive. In other words, codatatypes include infinite values, but not partial values. Thus support for codatatypes allows a larger class of Haskell programs to be formalized, although not as many as in HOLCF.

Coq uses a dependently-typed logic, which is more expressive than the simply-typed logic implemented in Isabelle and other HOL provers. The programming language ML uses a type system similar to Isabelle's, so HOLCF can express most all ML datatypes and programs; but Haskell has a richer type system intermediate between Isabelle and Coq. As a result, Haskell programs that use certain type system features cannot be formalized in HOLCF. Perhaps the most painful restriction is Isabelle's inability to express higher-order types, such as monad transformers, which are parameterized by other type constructors. (Note that the HOLCF formalization of the resumption monad transformer described earlier in this chapter used a fixed inner monad. Having the inner monad as a type parameter would be preferable—and such higher-order types are perfectly representable in the HOLCF '11 universal domain—but unfortunately such types are not expressible in Isabelle.) Isabelle's type system also rules out higher-rank polymorphism, existential datatypes, and nested datatypes [BM98], even though these could also be modeled in the universal domain. In contrast, Coq would have no problem expressing programs with any of these kinds of types. In summary, neither Coq nor HOLCF '11 is clearly more expressive than the other: Each can directly formalize some functional programs that the other cannot.

Coq and Isabelle have different approaches to automation. In Coq, users have a selection of numerous proof tactics with small, precise, predictable effects. Accordingly, Coq proof scripts often tend to contain many explicit details. On the other hand, Isabelle focuses more on just a few commonly-used tactics (particularly `induct`, `simp`, and `auto`) which are very powerful and extensible. Isabelle's simplifier (used by both `simp` and `auto`) played an important role in the development of HOLCF '11: Much of the proof automation in HOLCF '11 was implemented purely by choosing a set of carefully-formulated conditional rewrite rules to declare to the simplifier. Common subgoals involving admissibility, continuity, chains, comparisons, strictness, and more are all solvable by the simplifier. This is what makes the highly automatic induct-and-simplify proof scripts possible in HOLCF '11.

Each of the interactive theorem provers mentioned above is built on a small trusted proof kernel, inspired by the LCF-style architecture, so they provide a high-confidence argument for correctness. Perhaps HOL Light has a slight advantage over the others (including Isabelle) because it has the smallest, simplest proof kernel.

Language-specific interactive proof systems. Interactive proof tools exist that are specifically designed for reasoning about a specific programming language. Of these, the Sparkle theorem prover [MEP01] is the most closely related to HOLCF '11 in terms of its aims and its capabilities.

Sparkle is an interactive theorem prover that is custom-designed for reasoning about programs written in the lazy functional language Clean. Like HOLCF, it models language features like bottoms, partial and infinite values, strictness and laziness properties of functions, and strict and lazy datatypes. Datatype and function definitions are specified directly as programs written in (a subset of) the CLEAN language. Except for translating away a few unsupported language features, it is then almost trivial to import a CLEAN program into SPARKLE: The prover reasons directly on the source representation, so no translation process is necessary. (This is in contrast to HOLCF, which is intended to be able to express a common subset of various functional languages, but requires a bit of translation e.g. from Haskell.)

While the expressiveness of SPARKLE is very good for programs and datatypes, the formula language is more restricted. Properties in SPARKLE are expressed in a fixed formula language that includes equality, various logical connectives, and universal and existential quantification. User-defined predicates are limited to executable functions that produce booleans. Like the original LCF systems, it is first-order, so it is not possible to express induction principles as theorems—the hardwired induction tactic is the only form of induction available. (It is, however, possible to state and prove take lemmas.)

As for automation, SPARKLE has a collection of built-in proof tactics, including rewriting, induction over datatypes, case analysis, and many others. It also has a hint mechanism that can select and apply tactics automatically, so proofs of some simple theorems can completed without user interaction. In terms of its proof capabilities, SPARKLE can handle all the proofs about lazy lists that were shown in Sec. 7.2, although the automation has problems with functions like `repeat`, whose definitions can cause some rewriting strategies to loop.

SPARKLE is implemented in the CLEAN language. However, it does not adhere to the LCF architecture—to trust in the correctness of a proof, it is necessary to trust the code implementing each tactic used in the proof. Having to trust the implementation code of every tactic may have led to some tactics being a bit too conservative: For example, when performing induction over mutually inductive or indirect-recursive datatypes, the built-in induction principle is weaker than it could be, omitting induction hypotheses except for directly-recursive arguments.

Another tool that was partly inspired by Sparkle is the Haskell Equational Reasoning Assistant (HERA) [Gil06]. This tool provides a user-interface similar to Sparkle, where users can select subterms and apply a variety of rewriting-based tactics. In terms of proof capabilities, it is much more limited than Sparkle or LCF, since it supports only rewriting and not induction. It is not implemented in the LCF style, but HERA does record all of the equations used to rewrite a program, making it possible (in principle) to replay or check a HERA proof in another tool.

7.6 CONCLUSION

The research effort of developing HOLCF can be divided roughly into three areas of work: 1) formalizing domain theory and proving libraries of theorems; 2) designing proof heuristics and automation, configuring the simplifier and other tactics; and 3) implementing definition packages. All three items are tightly interdependent, of course. Choices about which domain-theoretic concepts to formalize are constrained by the requirements of the definition packages, and also influenced by the desire for better automation. Definition packages must generate theorems that implement proof automation for users; conversely, the definition packages are also users of proof automation themselves, while they are performing internal proofs.

In terms of choosing which domain-theoretic concepts to formalize in HOLCF, the primary lesson learned is the importance of keeping things small: To get a high level of proof automation, it is necessary to have library lemmas relating all possible combinations of core concepts. Formalizing all the definitions contained in a typical domain theory textbook would have been a disaster, because without an astronomical number of lemmas, there would necessarily be gaps in the theorem library, causing automation to suffer.

One concept that did *not* make it into HOLCF '11 was continuity on a set. Some functions in HOLCF are continuous only on a subset of their domains, e.g. Abs functions generated by type definitions with Cpodef. One possible design choice would have been to introduce, alongside the ordinary cont, a new notion of continuity on a set. The problem is that then every new function with a continuity rule also needs a separate rule for set-restricted continuity. This increases the size of the theory library, and also adds work for the definition packages, which must generate new lemmas of this kind. So instead of formalizing a new variation of continuity, we express continuity on a set in terms of ordinary continuity of composed functions (see e.g. lemma typedef_cont_Abs in Sec. 2.3). Overall, HOLCF seems to be better off without this extra variant of continuity.

One concept that *was* added to HOLCF '11 is compactness: In terms of admissibility proofs, compactness brought significant improvements in proof automation. But what was the full cost of integrating this new notion into HOLCF? Many new theorems had to be added to the library, relating compactness to admissibility, chain-finite types, type definitions, and various data constructor functions. Luckily compactness does not have notable interactions with very many other concepts, so the number of new theorems needed was not too large. In addition to the library lemmas, it was also necessary to have the Cpodef and Domain packages generate compactness lemmas for each newly-defined type. Fortunately the library lemmas about compactness made these new theorems relatively easy to produce, without requiring too much implementation code. Overall, considering the implementation effort versus the benefits, adding compactness can be considered a worthwhile improvement for HOLCF '11.

Perhaps the most important and far-reaching design decision in HOLCF '11 was the selection of the ω-bifinites as the preferred category of domains. We are fortunate to have found such a category that allows so many components to work well together: the powerdomain library, ideal completion, the universal domain, algebraic deflations, and the Domain package. Domain theory researchers—including Scott, Plotkin, and Gunter among others—have explored a variety of different categories of domains in recent decades [Plo76, GS90, AJ94, Sco08]. One of the primary motivations for this line of research is to identify categories that are suitable for modeling recursive datatypes and programs. So in some sense, the implementation of HOLCF '11 and the Domain package using the category of ω-bifinite domains is a realization of this research goal: By demonstrating a completely implemented, formal, executable model of recursive datatypes, we validate the ω-bifinite domains as a suitable category of domains for reasoning about computation.

As a final conclusion, we review some high-level characteristics of HOLCF '11 and what it is best used for. HOLCF '11 is not the only possible choice for doing interactive proofs about functional programs; indeed, in some situations other systems might be preferable. For example, for verification of programs that only manipulate finite values and terminate on all inputs, higher-order logic is probably a better choice—although HOLCF can certainly reason about terminating programs, the bottoms and partial values tend to get in the way. But in other situations, the unique properties of HOLCF '11 make it the best system available.

For reasoning about programs whose termination is difficult to establish, the support for general recursion in HOLCF is particularly valuable. Other programs require HOLCF due to their use of datatypes: Recursive definitions involving powersets or the full function space are not allowed in HOL or other logics of total functions; yet recursive datatypes involving powerdomains or the continuous function space are easy to define in HOLCF '11. While HOL might work well for self-contained programs whose inputs are known to be total and finite, HOLCF '11 is much more useful for verifying open-ended libraries that might be used with arbitrary inputs, finite or infinite, partial or total. In this situation, it is good to know that HOLCF '11 models the source programs and datatypes very accurately:

Datatypes include all the same partial and infinite values, and programs satisfy all the same laziness/strictness properties in HOLCF '11 as they do in Haskell.

When used for the kind of functional program verification it was designed for, HOLCF '11 is an effective and useful tool. First, the system makes it easy to translate a variety of functional programs into the formalism of the logic. HOLCF '11 then helps to make proofs about those programs as easy as possible: Using a combination of lemmas from libraries together with automatically-generated theorems, the proof automation in HOLCF '11 helps to discharge easy subgoals, letting users focus their attention on only the most challenging and interesting parts of proofs. Finally, the definitional implementation of HOLCF '11 means that users have a good reason to trust that their proofs are correct.

REFERENCES

[AC98] Roberto M. Amadio and Pierre-Louis Curien. *Domains and Lambda-Calculi*. Cambridge University Press, New York, NY, USA, 1998.

[Age94] Sten Agerholm. *A HOL Basis for Reasoning about Functional Programs*. PhD thesis, University of Aarhus, BRICS Department of Computer Science, 1994.

[AJ94] 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.

[Bal10] Clemens Ballarin. Tutorial to locales and locale interpretation. In L. Lambán, A. Romero, and J. Rubio, editors, *Contribuciones Científicas en honor de Mirian Andrés*, pages 1–17, Logroño, Spain, 2010.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall PTR, 2nd edition, May 1998.

[BKV09] Nick Benton, Andrew Kennedy, and Carsten Varming. Some domain theory and denotational semantics in Coq. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs '09)*, volume 5674 of *LNCS*. Springer, 2009.

[BM98] Richard S. Bird and Lambert G. L. T. Meertens. Nested datatypes. In *MPC '98: Proceedings of the Mathematics of Program Construction*, pages 52–67, London, UK, 1998. Springer-Verlag.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice Hall, 1 edition, 1988.

[BW99] Stefan Berghofer and Markus Wenzel. Inductive datatypes in HOL - lessons learned in formal-logic engineering. In *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLS '99)*, pages 19–36, London, UK, 1999. Springer-Verlag.

[DGHJ06] Nils Anders Danielsson, Jeremy Gibbons, John Hughes, and Patrik Jansson. Fast and loose reasoning is morally correct. In *POPL '06: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 206–217. ACM, Jan 2006.

[GH05] Jeremy Gibbons and Graham Hutton. Proof methods for corecursive programs. *Fundamenta Informaticae Special Issue on Program Transformation*, 66(4):353–366, April–May 2005.

[Gil06] Andy Gill. Introducing the Haskell Equational Reasoning Assistant. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, pages 108–109. ACM Press, 2006.

[Gim96] Eduardo Giménez. An application of co-inductive types in Coq: Verification of the alternating bit protocol. In *Selected papers from the International Workshop on Types for Proofs and Programs*, TYPES '95, pages 135–152, London, UK, 1996. Springer-Verlag.

[GMW79] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer-Verlag, 1979.

[Gor00] Mike Gordon. From LCF to HOL: a short history. In *Proof, Language, and Interaction*, pages 169–185, Cambridge, MA, USA, 2000. MIT Press.

[GS90] Carl A. Gunter and Dana S. Scott. Semantic domains. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B)*, pages 633–674. MIT Press, 1990.

[Gun85] Carl Gunter. *Profinite Solutions for Recursive Domain Equations*. PhD thesis, University of Wisconsin at Madison, 1985.

[Gun87] Carl A. Gunter. Universal profinite domains. *Information and Computation*, 72(1):1–30, 1987.

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

[Gun94] Elsa L. Gunter. A broader class of trees for recursive type definitions for HOL. In *Higher Order Logic Theorem Proving and Its Applications*, volume 780 of *LNCS*, pages 141–154. Springer, 1994.

[Haf10] Florian Haftmann. Haskell-style type classes with Isabelle/Isar. `http://isabelle.in.tum.de/doc/classes.pdf`, June 2010.

[HG01] Graham Hutton and Jeremy Gibbons. The generic approximation lemma. *Information Processing Letters*, 79(4):197–201, August 2001.

[HMW05] Brian Huffman, John Matthews, and Peter White. Axiomatic constructor classes in Isabelle/HOLCF. In Joe Hurd and Tom Melham, editors, *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs '05)*, volume 3603 of *LNCS*, pages 147–162. Springer, 2005.

[HSH02] William Harrison, Tim Sheard, and James Hook. Fine control of demand in Haskell. In *6th International Conference on the Mathematics of Program Construction, Dagstuhl*, pages 68–93. Springer, 2002.

[Hud98] Paul Hudak. Modular domain specific languages and tools. In *Proceedings of the Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society, June 1998.

[Huf08] Brian Huffman. Reasoning with powerdomains in Isabelle/HOLCF. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher-Order Logics (TPHOLs 2008): Emerging Trends*, pages 45–56, 2008.

[Huf09a] Brian Huffman. A purely definitional universal domain. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs '09)*, volume 5674 of *LNCS*, pages 260–275. Springer, 2009.

[Huf09b] Brian Huffman. Stream fusion. In Gerwin Klein, Tobias Nipkow, and Lawrence Paulson, editors, *The Archive of Formal Proofs*. `http://afp.sf.net/entries/Stream-Fusion.shtml`, April 2009. Formal proof development.

[Kra10a] Alexander Krauss. Partial and nested recursive function definitions in higher-order logic. *Journal of Automated Reasoning*, 44(4):303–336, April 2010.

[Kra10b] Alexander Krauss. Recursive definitions of monadic functions. In *Workshop on Partiality and Recursion (PAR-10)*, 2010.

[KWP99] Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson. Locales: A sectioning concept for Isabelle. In *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs '99)*, volume 1690 of *LNCS*, pages 149–166. Springer-Verlag, 1999.

[Lan66] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.

[LHJ95] Sheng Liang, Paul Hudak, and Mark P. Jones. Monad transformers and modular interpreters. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, CA, January 1995.

[Mel89] Thomas F. Melham. Automating recursive type definitions in higher order logic. In *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer-Verlag, 1989.

[MEP01] Maarten De Mol, Marko Van Eekelen, and Rinus Plasmeijer. Theorem proving for functional programmers – Sparkle: A functional theorem prover. In *The 13th International Workshop on Implementation of Functional Languages, IFL 2001, Selected Papers*, volume 2312 of *LNCS*, pages 55–72. Springer, 2001.

[MNOS99] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oskar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.

[Mog89] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, University of Edinburgh, June 1989.

[MP08] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

[Mül98] Olaf Müller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory.* PhD thesis, Technische Universität München, 1998.

[NP05] Tobias Nipkow and Lawrence C. Paulson. Proof pearl: Defining functions over finite sets. In Joe Hurd and Tom Melham, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2005)*, volume 3603 of *LNCS*, pages 385–396. Springer, 2005.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Ohe97] David von Oheimb. *Datentypspezifikationen in Higher-Order LCF.* PhD thesis, Technische Universität München, 1997.

[Pap01a] Nikolaos S. Papaspyrou. A resumption monad transformer and its applications in the semantics of concurrency. In *Proceedings of the 3rd Panhellenic Logic Symposium*, Anogia, Greece, July 2001.

[Pap01b] Nikolaos S. Papaspyrou. A resumption monad transformer and its applications in the semantics of concurrency. Technical Report CSD-SW-TR-2-01, National Technical University of Athens, School of Electrical and Computer Engineering, Software Engineering Laboratory, 2001.

[Pau84] Lawrence Paulson. Deriving structural induction in LCF. In *Proceedings of the international symposium on Semantics of data types*, pages 197–214, New York, NY, USA, 1984. Springer-Verlag New York, Inc.

[Pau87] Lawrence C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF.* Cambridge University Press, New York, NY, USA, 1987.

[Pau97] Lawrence C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *Journal of Logic and Computation*, 7, 1997.

[Pit94] A. M. Pitts. A co-induction principle for recursively defined domains. *Theoretical Computer Science*, 124:195–219, 1994. (A preliminary version of this work appeared as Cambridge Univ. Computer Laboratory Tech. Rept. No. 252, April 1992.).

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[PJ03] Simon Peyton Jones. Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 2003.

[PJRH$^+$99] Simon Peyton Jones, Alastair Reid, Fergus Henderson, Tony Hoare, and Simon Marlow. A semantics for imprecise exceptions. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 25–36, New York, NY, USA, 1999. ACM.

[Plo76] Gordon D. Plotkin. A powerdomain construction. *SIAM J. Comput.*, 5(3):452–487, 1976.

[PM00] 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.

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

[Reg95] 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*, volume 971 of *LNCS*, pages 293–307, Aspen Grove, Utah, 1995. Springer-Verlag.

[Sco93] Dana S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121:411–440, 1993.

[Sco08] Dana Scott. Semilattices and domains. Workshop Domains IX, Brighton, UK, `http://www.informatics.sussex.ac.uk/events/domains9`, September 2008.

[Sli96] Konrad Slind. Function definition in higher-order logic. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Joakim von Wright, Jim Grundy, and John Harrison, editors, *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics (TPHOLs '96)*, volume 1125 of *LNCS*, pages 381–397. Springer, 1996.

[Tel04] Amber Telfer. Fixrec. Master's thesis, Oregon Health and Science University, OGI School of Science and Engineering, 2004.

[Thi95] Peter Thiemann. Towards a denotational semantics for concurrent state transformers. In Masato Takeichi, editor, *Fuji Workshop on Functional and Logic Programming*, pages 19–33, Fuji Susono, Japan, July 1995. World Scientific Press, Singapore.

[Wad87] Philip Wadler. Efficient compilation of pattern-matching. In Simon L. Peyton Jones, editor, *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*, chapter 5, pages 78–103. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[WW07] Makarius Wenzel and Burkhart Wolff. Building formal method tools in the Isabelle/Isar framework. In Klaus Schneider and Jens Brandt, editors, *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs '07)*, volume 4732 of *LNCS*, pages 352–367, Berlin, Heidelberg, 2007. Springer-Verlag.

Appendix A

INDEX OF ISABELLE DEFINITIONS

Note: Entries marked with an asterisk (*) are specific to this document; all other entries can also be found in the Isabelle/HOLCF 2011 sources.

Appendix B

INDEX OF ISABELLE THEOREMS

Note: Entries marked with an asterisk (*) are specific to this document; all other entries can also be found in the Isabelle/HOLCF 2011 sources.