Thesis Proposal

Brian Huffman

December 23, 2009

Abstract

The goal of my thesis project is to create an improved framework for reasoning about functional programs in the Isabelle theorem prover. Instead of reasoning about syntax, functional programs will be modeled directly in terms of their denotational semantics, using various concepts from domain theory. Confidence in the framework will be ensured by adhering to a definitional approach: New constants and types must be defined in terms of previous concepts, without introducing new axioms.

Existing theorem prover tools do not adequately support reasoning about functional programs that use general recursion and recursive datatypes. My thesis statement is that the approach described here will permit proofs of correctness for a variety of real-world functional programs that are beyond the scope of current tools.

1 Introduction

How can we ensure the correctness of programs and libraries written in a lazy functional programming language like Haskell? To achieve the highest level of assurance, we want to use formal reasoning—theorem provers and other tools for constructing computer-checked proofs.

Recursive definitions are a central feature of functional programming languages. In Haskell, recursion appears both at the value level (recursively-defined functions) and at the type level (algebraic datatypes). My thesis project involves the creation of tools to automate the formalization of both kinds of recursive definitions.

The Higher Order Logic of Computable Functions (Isabelle/HOLCF) is a formalization of domain theory for the Isabelle theorem prover, implemented as an extension to Higher Order Logic (Isabelle/HOL). It is designed for doing formal reasoning about functional programs. However, the usefulness of HOLCF has been relatively limited. Until recently, it lacked automation for defining recursive functions. Even now, only a restricted subset of recursive type definitions are well-supported. Also, HOLCF is not a conservative extension of Isabelle/HOL—some kinds of definitions are effected by generating new axioms, which introduces concerns about the possibility of unsoundness. All of these issues restrict the set of programs that HOLCF can reason about.

My thesis statement is that HOLCF can be extended to facilitate formal reasoning about useful functional programs that can not be reasoned about in existing theorem provers. Furthermore, the new HOLCF can be implemented using only conservative extensions to Higher Order Logic—no new axioms are required.

To support my thesis statement, I will need to:

- Complete the implementations of the recursive definition packages
- Perform case studies to demonstrate the feasibility of using the tools for real proof applications

2 Thesis Outline

Each subsection listed here corresponds to one chapter of the proposed thesis document. For each chapter, I give a short description of its contents, focusing on the particular technical problems whose solutions will be written up in each chapter.

1. Introduction
2. HOLCF and Basic Domain Theory
3. Recursive Value Definitions: The `fixrec` Package
4. Basic HOLCF Types: Sums, Products, Functions, and Lifting
5. Recursive Datatype Definitions: The `domain` Package
6. Advanced HOLCF Types: Powerdomains and Ideal Completion
7. Representing Types using Deflations
8. Alternative Representations for Types
9. Constructing the Universal Domain
10. Building a Definitional Domain Package
11. Case Studies
12. Conclusions and Future Work

2.1 Introduction

This section will talk about some motivating examples, i.e. what kinds of Haskell programs are we interested in reasoning about. Also, it will show that even simple proofs about lazy functional programs (e.g. ones using induction) can have subtle side conditions that are easy to get wrong in a pencil-and-paper

proof. This supports the idea that it is useful and worthwhile to develop tools for formal, computer-checked reasoning about lazy functional programs.

The introduction will also give a short history of research in this area, starting with the domain-theoretic models of the untyped lambda-calculus by Dana Scott, continuing through work on the LCF theorem prover by Robin Milner and others, and finally leading up to more recent systems for reasoning about higher-order functional programs, including HOLCF.

2.2 HOLCF and Basic Domain Theory

Domain theory is a branch of mathematics that is very useful for modeling functional programming languages. This chapter will explain some basic concepts in domain theory, and show why they are useful for modeling Haskell types and programs. It will also contain an overview of the basic parts of HOLCF (Higher-Order Logic of Computable Functions), a formalization of domain theory in the Isabelle theorem prover.

What is the mathematical meaning of a Haskell type? We can start by assigning a denotation (i.e. a mathematical meaning) to each Haskell term. Then Haskell types can be modeled simply as sets of denotations, but they really have more structure than that.

As Haskell is a lazy language, a Haskell term is not evaluated until its value is needed. So for any given type, there is always a possibility that a Haskell term of that type may fail to terminate when evaluated; such a term is denoted by the *bottom element* of its type, written $\bot$. Such a value is also called *undefined*.

With lazy data structures, it is also possible to have *partial values*. For example, a pair of type `(Bool, Int)` may have a first element that evaluates to `True`, but evaluating the second element might fail to terminate. Such a list would be denoted by $(\mathit{True}, \bot)$. Function types also contain partial values, where the function terminates on some inputs but not others.

Partial values give rise to an ordering relation, where some values are *less defined* than others. This relation is written as $(\sqsubseteq)$. For example:

$$\bot \sqsubseteq (\bot, \bot) \sqsubseteq (\mathit{True}, \bot) \sqsubseteq (\mathit{True}, 5)$$

In addition to partial values, recursive datatypes in Haskell may also include *infinite* values. For example, in Haskell we can define an infinite list value `trues` of type `[Bool]` by writing `trues = True : trues`. In the domain ordering $(\sqsubseteq)$, the denotation of `trues` is the least upper bound of the sequence of finite approximations:

$$\bot \sqsubseteq \mathit{True} : \bot \sqsubseteq \mathit{True} : \mathit{True} : \bot \sqsubseteq \ldots \sqsubseteq \mathit{trues}$$

A set with a partial order $(\sqsubseteq)$ that has least upper bounds for all increasing sequences is called a *complete partial order*, or *cpo*. Any Haskell type can be modeled by a cpo.

It also turns out that all functions definable in Haskell are *continuous*, which means that they preserve cpo structure, mapping least upper bounds to least

upper bounds. Haskell function types can thus be modeled most precisely using the continuous function space between two cpos.

Modeling Haskell types as cpos instead of just sets is useful because the cpo structure allows us to define recursive values. For any cpo A, and continuous function f of type $A \to A$, we can find a *fixed point*, i.e. a value x such that $f(x) = x$.

To formalize the process of finding fixed points, HOLCF defines a *least fixed-point combinator*:

$$\mathit{fix}(f) = \bigsqcup_n f^n(\bot)$$

The least fixed-point combinator satisfies the property $f(\mathit{fix}(f)) = \mathit{fix}(f)$. The next chapter about the `fixrec` package will describe how the combinator can be used to simulate recursive definitions.

2.3 Recursive Value Definitions: The `fixrec` Package

Recursive function definitions are a built-in part of Haskell. How can we model these in a theorem prover?

Any recursive function or value definition in Haskell can be rewritten as a *non*-recursive definition using the fixed-point combinator `fix`. For example, here is the recursive value `trues` from the previous section:

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

trues :: [Bool]
trues = fix (\rec -> True : rec)
```

For recursive definitions that use pattern matching, we need to convert the patterns into a case expression. Here is an example:

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

sum :: [Int] -> Int
sum = fix (\rec zs -> case zs of [] -> 0; x : xs -> x + rec xs)
```

In HOLCF, the *fixrec* package automates this process of converting a Haskell-style recursive definition with patterns into a simple definition using *fix*. This thesis chapter will describe how pattern matching is formalized in HOLCF. It will also contain a full description of how the *fixrec* package works, including pattern-match compilation and automatic generation of induction rules.

2.4 Basic HOLCF Types: Sums, Products, Functions, and Lifting

Functional languages like Haskell provide a wide range of definable types, each of which can be modeled as a complete partial order (cpo). To construct all of these various cpos in a theorem prover is a significant undertaking—most of the remainder of the dissertation will be devoted to this task. As a first step, this chapter will describe how to formalize a few basic cpos, including sums, products, function spaces, and lifted types.

Some cpos used in HOLCF are introduced as ordinary types in Isabelle/HOL, and should be familiar to anyone: the set-theoretic function space $A \Rightarrow B$, the Cartesian product $A \times B$, and the disjoint sum $A + B$.

In addition to these, HOLCF introduces a few new types: The *continuous function space* $A \to B$ is a subtype of the full function space $A \Rightarrow B$, consisting of only the continuous functions from A to B. The *lifted type* $A_\bot$ consists of a copy of A adjoined with a single new element, $\bot$. The *strict sum* $A \oplus B$ (also known as the coalesced sum) is like the disjoint sum, except that the bottom elements from A and B are identified—this means the injection functions $sinl : A \to A \oplus B$ and $sinr : B \to A \oplus B$ are strict functions, i.e. $sinl(\bot) = sinr(\bot) = \bot$. Similarly, the *strict product* $A \otimes B$ is like the cartesian product, except the pair constructor is strict in both arguments. The lifted type, strict sum and strict product model the following Haskell types:

```
data Lift a = Up a  -- lazy constructor
data StrictSum a b = InL !a | InR !b
data StrictProduct a b = Pair !a !b
```

Some cpos are defined in HOLCF by a completely manual process: Starting with an existing type, we can define an ordering relation, and then proceed to prove that the ordering is a complete partial order. For example, the ordering relation on the product type $A \times B$ is defined component-wise, so that $(a, b) \sqsubseteq (c, d) \iff a \sqsubseteq c \wedge b \sqsubseteq d$. It is then proved that if A and B are cpos, then this product ordering makes $A \times B$ into a cpo. The cpo instances for $A \Rightarrow B$, $A + B$, $A_\bot$, and discrete cpos like *unit* and *bool* are constructed and proved in a similar way.

The process of proving the cpo axioms manually is generally 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 common way to construct a new cpo is to carve out a subset of an existing cpo; then the new cpo can inherit the order structure from the old cpo. This makes the proofs much easier than defining the cpo from scratch. It is automated in HOLCF using the `cpodef` package. Using `cpodef` saves a lot of proof effort, when it is possible to use it. This chapter will contain a full description of how `cpodef` works, and what proof obligations are required.

The continuous function space $A \to B$ is defined using `cpodef` as the set $\{f : A \Rightarrow B \mid cont(f)\}$; it inherits its ordering from the set-theoretic function space. Similarly, the strict product $A \otimes B$ is defined as a sub-cpo of type $A \times B$.

The strict sum is also defined using `cpodef`, using a slightly more complex encoding that will be described in the thesis chapter.

2.5 Recursive Datatype Definitions: The `domain` Package

Datatype definitions are a standard feature of the Haskell programming language. Datatypes are specified as a *sum of products*: The programmer gives a list of constructors, each of which has a list of argument types. Here is an example of a simple datatype:

```
data OneTwo a = One a | Two a a
```

Non-recursive datatypes like this one can be modeled in HOLCF by a simple combination of the strict sum, strict product and lifting type constructors introduced in the previous section. For example, if A is a cpo that models the Haskell type `a`, then $A_\perp \oplus (A_\perp \otimes A_\perp)$ is a model of type `OneTwo a`. (The lifting is necessary because the constructor arguments are lazy; omitting the lifting would yield a strict version.)

Haskell also allows datatype definitions to be *recursive*, where the type being defined may appear on the right-hand side of the definition. Here is an example of a binary tree datatype; the branch constructor takes other trees as arguments, making this a recursive definition.

```
data Tree = Leaf Int | Branch Tree Tree
```

How can we use sums and products to specify more complicated recursive types? The solution in HOLCF is to use *domain isomorphisms*. If we want to model the `Tree` datatype in HOLCF, we need a pair of continuous functions that form an isomorphism:

$$\begin{aligned} \mathit{repTree} &:: \mathit{Tree} \to \mathit{Int}_\perp \oplus (\mathit{Tree}_\perp \otimes \mathit{Tree}_\perp) \\ \mathit{absTree} &:: \mathit{Int}_\perp \oplus (\mathit{Tree}_\perp \otimes \mathit{Tree}_\perp) \to \mathit{Tree} \end{aligned}$$

Using these functions (together with the constructors and case-analysis operations for sums, products, and lifting) we can define the constructors and case-analysis combinator for the *Tree* type. If *repTree* and *absTree* are an isomorphism, then we can prove that the constructors *Leaf* and *Tree* are distinct and exhaustive, and that the case-analysis combinator obeys the rules we would expect.

In HOLCF, the *domain package* automates the process of declaring a domain isomorphism, defining constructor functions and case combinators, and proving basic properties about them. In addition, the domain package also declares a suitable induction rule for each newly-defined datatype.

However, as it is currently implemented, the domain package takes some serious shortcuts. Instead of defining the domain isomorphisms explicitly, it specifies the *rep* and *abs* functions axiomatically, simply assuming that they form an isomorphism. The induction rule is similarly axiomatized.

Having a theorem-proving tool that generates new axioms on-the-fly is very problematic. Theorem provers like Isabelle are designed to have a small trusted kernel; anyone can write code that builds theorems using the kernel operations, and the theorems are guaranteed to be correct. However, by adding new axioms, the domain package adds a lot more code that needs to be trusted. A bug in this code can cause the prover to be unsound.

Most of the later thesis chapters are focused 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 section focuses instead on another language feature: non-determinism. Along the way, we will develop some infrastructure which will come in handy for implementing a definitional domain package, among other things.

2.6 Advanced HOLCF Types: Powerdomains and Ideal Completion

At first glance, it may not seem that non-determinism would be necessary for reasoning about functional programs—after all, in the absence of side-effects, the evaluation of purely functional computations is deterministic. However, real-world languages like Haskell also include I/O operations that may have non-deterministic side-effects. Another place where non-determinism can pop up is multi-threaded programs, where there may be many possible ways to interleave the execution of the various threads.

In set theory, we can use the *powerset* of a type to model computations that have multiple possible values. For example, we can use the set $\{3, 4, 5\}$ to denote a non-deterministic computation of type integer that has those three possible return values. A *powerdomain* is the domain-theoretic version of this concept: The powerdomain operations for creating singletons and taking unions are *continuous functions*. Therefore powerdomains can model computations that combine non-determinism with laziness and general recursion.

Formalizing powerdomains in HOLCF will require some new techniques. The previous section on basic HOLCF types mentioned that some cpos are defined as subset of previous cpos, using *cpodef*. 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*.

To define a cpo, we start by defining its *basis*, which consists of only the *finite* elements of the cpo. The basis only needs to be a partial order, not necessarily a complete partial order. The operations on the basis only need to be monotone, not necessarily continuous. (Monotonicity is generally much easier to prove than continuity.) The ideal completion process extends the basis with new infinite elements to give a cpo; a similar process 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; this thesis chapter will describe the formalization, and show how it is used to define the

powerdomain type constructors. The process is general enough to be useful for other cpos besides powerdomains; a later chapter will show how it is used to construct a universal domain.

The powerdomain construction does not work for just any arbitrary cpo; the element type must be an *algebraic* cpo, i.e. it must have a *basis*. For this reason, it is necessary to define a more restrictive class of algebraic cpos for the ideal completion library to work with.

To support the ideal completion library, I formalize the *omega-bifinite domains*. An omega-bifinite domain is a cpo that is the least upper bound (in some sense) of some sequence of finite cpos. There are various technical reasons why this was a good design choice for HOLCF, but the main reasons are as follows:

- Omega-bifinite domains have a basis of finite elements.
- Omega-bifiniteness is preserved by all type constructors in HOLCF, including powerdomains.
- All recursive datatypes definable in Haskell are omega-bifinite.

This thesis chapter will describe how the omega-bifinite type class is formalized in HOLCF, and show how the omega-bifinite structure is used by the ideal completion library.

2.7 Representing Types using Deflations

Given a datatype definition, how can we construct a cpo that satisfies the appropriate domain isomorphism? We want to avoid declaring axioms. So to construct a domain isomorphism, we need to define the new cpo type as a subset of some pre-existing type using Isabelle's *typedef* command. We are left with two questions: First, what pre-existing type to use? And second, how to define an appropriate subset of that type?

My chosen solution is to use a single *universal domain* as the pre-existing type, and to use *deflations* to specify subsets of that type. This chapter will talk about deflations and related concepts. Specifics about the universal domain will be covered in a later chapter.

A *deflation* over a cpo C is a special kind of continuous function f of type $C \to C$ that is idempotent, so that $f \circ f = f$. The image of a deflation over C identifies a subset of C that is a cpo.

Some additional properties of deflations make them very nice for defining recursive datatypes: The set of all deflations over a cpo C is itself a cpo with a bottom element. This means that we can use the least fixed-point combinator *fix* to define deflations. Furthermore, if C is a suitable universal domain, then we can represent type constructors like sums, products, and lifting by *continuous functions*. As continuous functions, any domain equation built from these type constructors will be guaranteed to have a solution, given by *fix*.

Essentially, a deflation is a way to represent an Isabelle *type* as a *function value*. HOLCF has a lot of existing infrastructure for defining recursive

functions; deflations make it possible to reuse this infrastructure for defining recursive *types*.

In addition to deflations, this thesis chapter will also talk about the related concept of embedding-projection pairs (for embedding other types into a universal domain). It will also give details about how the continuous functions representing type constructors are defined. Finally, it will explain how to use a deflation to actually define a new cpo type and construct a domain isomorphism.

2.8 Alternative Representations for Types

Deflations are not the only possible representation for Haskell types. This chapter will describe a few alternatives to deflations, explain the benefits and tradeoffs of each, and show why deflations are a good choice for representing Haskell types in the Isabelle theorem prover.

The leading alternatives to the deflation model are based on partial equivalence relations (PERs); like the deflation model, they make use of a universal domain to represent Haskell values. Another variation of a PER model could use a type of syntactic terms, rather that a universal domain of values.

2.9 Constructing the Universal Domain

How can we define a universal omega-bifinite domain in Isabelle?

Constructions of a universal omega-bifinite domain exist in the domain theory literature. This chapter will show how to adapt one such construction so that it can be formalized in a theorem prover like Isabelle. The formalization makes use of the ideal completion process described in an earlier section.

This chapter will also include a section about how to represent deflations over this universal domain. Every deflation over an omega-bifinite cpo identifies a subset that is a cpo. However, the subset is not necessarily an *omega-bifinite* cpo. Therefore we must restrict our attention to only those deflations that represent omega-bifinite cpos—the so-called *algebraic* deflations. The thesis chapter will show how to define a type of algebraic deflations, and how to define operations on them.

2.10 Building a Definitional Domain Package

How can the domain package be modified to avoid generating new axioms?

Most of the existing domain package can be reused as is. In particular, the code for defining constructors, case analysis combinators, and related functions and proving properties about them will remain unchanged. Mostly it will be a matter of adding new code that defines the new cpo and domain isomorphism functions, and proves the theorems that were previously asserted as axioms.

What new features will the domain package have? Any recursive datatypes that were fully supported before will continue to work, and all the same theorems and induction rules will be proved, but without generating new axioms.

Additionally, datatypes definitions that use indirect recursion, which were not fully supported before, will work to some extent. The `Tree` datatype below is an example of indirect recursion, since a recursive occurrence of the `Tree` type is wrapped inside the pre-existing `List` type constructor.

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

data Tree a = Leaf a | Branch (List (Tree a))
```

For such definitions, the domain package will prove the right reach lemma (a primitive form of induction rule); however, it will not generate nice induction rules with separate cases for each constructor. Users can derive induction rules for such datatypes manually from the reach lemma.

What features will be removed? The new domain package will be restricted so that only omega-bifinite domains can be defined. Similarly, type parameters will be constrained to the omega-bifinite class. I plan to preserve the original behavior of the domain package as a user-selectable option, in case the switch to the omega-bifinite class breaks any existing HOLCF applications.

2.11 Case Studies

I plan to have two case studies, which will make use of different new features of HOLCF. One will focus on the definition of recursive *values* using the *fixrec* package, and the other will focus on the definition of recursive *types* using the *domain* package.

The first case study will look at a formalization of *stream fusion* in HOLCF. Stream fusion is a reimplementation of Haskell's standard list library, designed to allow the compiler to eliminate intermediate data structures in list-transforming code. The library uses some interesting forms of recursion; the associated correctness proofs are a good demonstration of a technique called *least fixed-point induction*.

The second case study involves reasoning about some nontrivial datatypes that cannot be defined by traditional definition packages. In particular, I will examine some *monad* types of the kind often used in Haskell programming. Monads are typically used to implement computations with side-effects such as exceptions or mutable state; there are different monad types for 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. There are some standard laws ("the monad laws") that `return` and `bind` are expected to satisfy. 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 general approach with the second case study is to define a series of increasingly sophisticated monad datatypes, combining various kinds of side-effects. For each monad, I will define the standard operations including `return`

and `bind`, and prove that the operations satisfy the appropriate laws. These proofs will often require customized induction principles to be derived for each type. To demonstrate the unique features of the new domain package, some of the monad types will use powerdomains or negative recursion (i.e. recursion on the left of a function arrow). Existing type definition packages in other theorem provers support neither of these features.

2.12 Conclusions and Future Work

The conclusion chapter will summarize the contributions of the work, and also preview some directions for future research. One question reserved for future work is how to generate nice induction rules for indirect-recursive datatypes, like the `Tree` type mentioned earlier. In some cases it is possible to treat indirect recursion like mutual recursion, which suggests a particular form of induction rule; however, other formulations of induction rules are also possible. Exploring this area is beyond the scope of my thesis.

Other directions for future work involve adding support for pattern matching and recursive let-bindings to HOLCF. I have done some initial work on how to *express* such concepts in HOLCF, so users can write case- and let-expressions having the correct meaning, using nice syntax. However, finding the best way to interactively *reason* with such expressions is a much deeper problem.

This section will also have a survey of related work. This will include other formalizations of domain theory, other implementations of definition packages for theorem provers, and alternative approaches (besides domain theory) to formalizing functional languages.

3 Pre-Existing Work

My thesis project builds upon the work of others.

The HOLCF library of domain theory was originally created in the mid-1990s by Franz Regensburger; it included formalizations of complete partial orders, continuous functions, least fixed-points, and a few basic type constructors. the thesis chapter "HOLCF and Basic Domain Theory" will primarily consist of a review of material formalized in the original HOLCF. Parts of another chapter, "Basic HOLCF Types: Sums, Products, Functions, and Lifting" will also cover some material present in the original HOLCF.

In the late 1990s, David von Oheimb introduced the first version of the domain package, for defining recursive datatypes. The thesis chapter "Recursive Datatype Definitions: The `domain` Package" will contain a technical description of this work, highlighting the parts that will be extended by my work.

In the mid-2000s, Amber Telfer created the *fixrec* package for defining recursive functions in HOLCF; new features have been added by myself over the past few years. A description of this package will be found in the thesis chapter "Recursive Value Definitions: The `fixrec` Package".

4 Relevant Publications

During my time as a graduate student, I have published a few papers that are relevant to my thesis work. I should be able to reuse some material from these papers for some of the later thesis chapters.

In 2005 John Matthews, Peter White and I published "Axiomatic Constructor Classes in Isabelle/HOLCF" [4]. The paper includes much material that will form part of thesis chapter "Representing Types using Deflations".

In 2008, I published "Reasoning with Powerdomains in Isabelle/HOLCF" [1]. This paper describes the use of ideal completion to construct powerdomains; its description of the formalization of ideal completion will be useful for thesis chapter "Advanced HOLCF Types: Powerdomains and Ideal Completion". It also describes the formalization of the bifinite type class in HOLCF, which will become another part of the same chapter.

Most recently, I published the conference paper "A Purely Definitional Universal Domain" in 2009 [2]. This paper contains all the material that will make up up thesis chapter "Constructing the Universal Domain".

Also in 2009, I contributed an entry to the online Archive of Formal Proofs, containing a formalization of stream fusion [3]. I have also written a draft paper describing this work, which will make up a significant piece of the "Case Studies" chapter.

5 Remaining Work and Timeline

An initial version of the definitional domain package has recently been completed. The remaining technical work consists primarily of doing the second case study, described previously.

Besides the remaining case study, most of the remaining work consists of writing the thesis document. But I also expect to spend a bit more time working on the implementation: While writing up, I expect to do some cleaning up and reorganizing of the code, to make it easier to explain. Also, some bug fixes to the code might be necessary, if any bugs are found that impede progress on the case study. An estimated timeline is shown below:

- Case study with recursive datatypes (2–3 weeks)
- Clean up domain package implementation (2–3 weeks)
- Writing (4–6 months)

The total estimated remaining time is in the range of 5–8 months, which places the target completion date in late spring or summer 2010.

References

[1] 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.

[2] Brian Huffman. A purely definitional universal domain. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Proc. 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs '09), LNCS 5674*, pages 260–275. Springer, 2009.

[3] 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.

[4] Brian Huffman, John Matthews, and Peter White. Axiomatic constructor classes in Isabelle/HOLCF. In Joe Hurd and Tom Melham, editors, *Proc. 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs '05), LNCS 3603*, pages 147–162. Springer, 2005.