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

Brian Huffman

Portland State University

July 13, 2011

PhD Thesis Committee: James Hook (chair)
John Matthews (advisor)
Mark Jones
Tim Sheard
Gerardo Lafferriere

The Problem

We write computer programs, and want to know they are correct

Focus: Pure functional programming languages

Haskell

- Based on typed lambda calculus
- Supports equational reasoning (due to purity)
- Arbitrary recursive definitions
- Good for embedding other languages

Haskell expressions and types

```
              3  ::  Int
            (+)  ::  Int -> Int -> Int
   (\x -> x + 3) ::  Int -> Int
     (\f -> f 3) ::  (Int -> a) -> a
(\f x -> f (f x)) :: (a -> a) -> a -> a
```

Verification example: List datatype

List datatype with map function

```
data List a = Nil | Cons a (List a)
  -- includes both finite and infinite lists

map :: (a -> b) -> List a -> List b
map f Nil = Nil
map f (Cons x xs) = Cons (f x) (map f xs)
```

Theorem

The map function preserves composition:

`∀f g xs. map f (map g xs) = map (\x -> f (g x)) xs`

Verification example: List datatype (informal proof)

Proof.

By induction on `xs`.

- Base case (`Nil`):
  Show `map f (map g Nil) = map (\x -> f (g x)) Nil`.

Verification example: List datatype (informal proof)

Proof.

By induction on `xs`.

- Base case (`Nil`):
  Show `map f (map g Nil) = map (\x -> f (g x)) Nil`.
- Inductive case (`Cons`):
  Assume `map f (map g xs) = map (\x -> f (g x)) xs`.
  Show `map f (map g (Cons x xs)) = map (\x -> f (g x)) (Cons x xs)`.

Verification example: List datatype (informal proof)

Proof.

By induction on `xs`.

- Base case (`Nil`):
  Show `map f (map g Nil) = map (\x -> f (g x)) Nil`.
- Inductive case (`Cons`):
  Assume `map f (map g xs) = map (\x -> f (g x)) xs`.
  Show `map f (map g (Cons x xs)) = map (\x -> f (g x)) (Cons x xs)`.
- Base case ($\perp$):
  Show `map f (map g ⊥) = map (\x -> f (g x)) ⊥`.

Verification example: List datatype (informal proof)

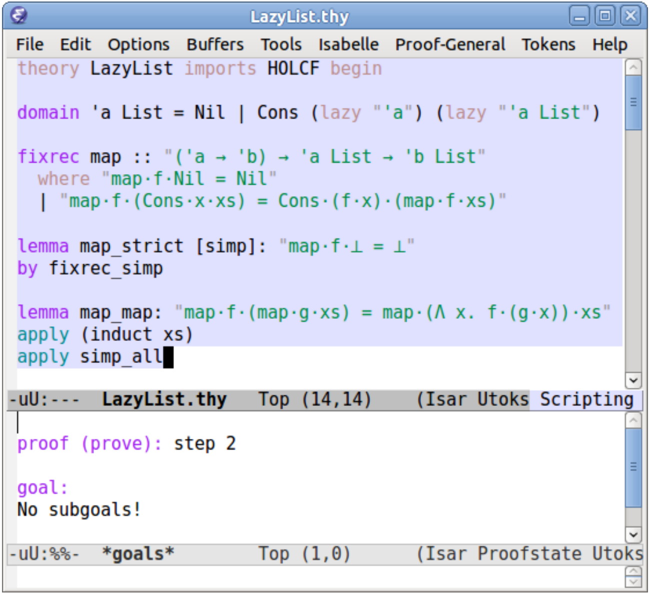
Proof.

By induction on `xs`.

- Base case (`Nil`):
  Show `map f (map g Nil) = map (\x -> f (g x)) Nil`.
- Inductive case (`Cons`):
  Assume `map f (map g xs) = map (\x -> f (g x)) xs`.
  Show `map f (map g (Cons x xs)) = map (\x -> f (g x)) (Cons x xs)`.
- Base case ($\perp$):
  Show `map f (map g `$\perp$`) = map (\x -> f (g x)) `$\perp$.
- Admissibility condition:
  Check that the goal is *admissible* in `xs`.

□

Verification example: List datatype (HOLCF proof)



Thesis statement

HOLCF-11 provides an unprecented combination of these qualities:

Expressiveness

- Can reason about a wide variety of programs
- Can formulate all kinds of properties

Automation

- Tools generate common useful theorems
- Tactics discharge commonly-occurring subgoals in proofs
- Easy proofs are automatic, hard proofs are possible

Confidence

- Theorems derived with primitive inferences from axioms of set theory
- Soundness ensured by construction

Harder example: Tree datatype

Tree datatype with map function

```
data Tree a = Leaf a | Node (List (Tree a))
  -- indirect recursion with list type

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f (Leaf x) = Leaf (f x)
mapTree f (Node ts) = Node (map (mapTree f) ts)
```

Theorem

The map function preserves composition:

```
∀f g t. mapTree f (mapTree g t) = mapTree (\x -> f (g x)) t
```

Harder example: Tree datatype (informal proof)

Proof.

Define abbreviation P(t) $\equiv$

```
mapTree f (mapTree g t) = mapTree (\x -> f (g x)) t.
```

Show P(t) by induction on `t`.

- Admissibility condition:
  Check that P(t) is admissible in `t`.
- Base case:
  Show P($\perp$).
- Base case:
  Show P(`Leaf x`).
- Inductive case:
  Fix arbitrary `h :: Tree a -> Tree a`; assume $\forall$`t.` P(`h t`).
  Show P(`Node (map h ts)`).

□

Harder example: Tree datatype (HOLCF proof)

Step 1: Derive induction rule

LazyList.thy

File Edit Options Buffers Tools Isabelle Proof-General Tokens Help

```
domain 'a Tree = Leaf (lazy "'a") | Node (lazy "'a Tree List")

lemma Tree_induct [induct type: Tree]:
  fixes P :: "'a Tree ⇒ bool"
  assumes adm: "adm P"
  assumes bottom: "P ⊥"
  assumes Leaf: "⋀x. P (Leaf·x)"
  assumes Node: "⋀f ts. ∀t::'a Tree. P (f·t) ⟹ P (Node·(map·f·ts))"
  shows "P t"
proof (rule Tree.take_induct)
  show "adm P" by fact
next
  fix n show "P (Tree_take n·t)"
  proof (induct n arbitrary: t)
-u-:---  LazyList.thy    40% (44,29)     (Isar Utoks Scripting )----------
```

Harder example: Tree datatype (HOLCF proof)

Step 2: Prove map theorem

LazyList.thy

File Edit Options Buffers Tools Isabelle Proof-General Tokens Help

```
fixrec mapTree :: "('a → 'b) → 'a Tree → 'b Tree"
  where "mapTree·f·(Leaf·x) = Leaf·(f·x)"
  | "mapTree·f·(Node·ts) = Node·(map·(mapTree·f)·ts)"

lemma mapTree_strict [simp]: "mapTree·f·⊥ = ⊥"
by fixrec_simp

lemma mapTree_mapTree:
  "mapTree·f·(mapTree·g·t) = mapTree·(Λ x. f·(g·x))·t"
apply (induct t)
apply (simp_all add: map_map)
done
```

-u-:--- LazyList.thy Bot (71,4) (Isar Utoks Scripting)----------

Background

Formalisms for programs and properties

Higher Order Logic (HOL)

- Based on typed lambda calculus
- Quantification over predicates (higher order)
- Has model in set theory

Logic of Computable Functions (LCF)

- Based on typed lambda calculus
- Has fixed-point combinator: $\mathrm{fix}(f) = f(\mathrm{fix}(f))$
- Every type has bottom element: $\perp$
- Has model in domain theory

Domain theory

Complete partial orders (cpos)

- partial ordering ($\sqsubseteq$)
- least element ($\perp$)
- every countable ascending sequence (chain) has least upper bound

Continuous functions

- preserve least upper bounds of chains
- continuous function space between cpos is a cpo

Least fixed-point operator

- $\mathrm{fix}(f) = f(\mathrm{fix}(f))$ for continuous f
- lub of chain: $\perp$, $f(\perp)$, $f(f(\perp))$, $f(f(f(\perp)))$, ...

Interactive theorem proving

Programmable systems for building formal proofs

- Every primitive logical inference checked by computer

LCF series of theorem provers (1970s–80s)

- Trusted kernel provides abstract "theorem" type
- Kernel operations $\Leftrightarrow$ logical inference rules
- Only the kernel can build theorems

HOL family of theorem provers (1980s–current)

- "LCF style" implementation gives high assurance
- Isabelle/HOL

Combining HOL and LCF

Higher Order Logic of Computable Functions (HOLCF)

- Model of LCF logic, constructed within Isabelle/HOL
- Domain theory built up from axioms of set theory
- Cpos and continuous functions coexist with ordinary types and functions
- Earlier versions (T.U. Munich, 1990s):
  - HOLCF-95 (basic domain theory, type constructors)
  - HOLCF-99 (introduces **domain** command)

Technical Contributions

Original work in three areas

Theory libraries

- Identify more useful concepts from domain theory literature
- Formalize and prove collections of theorems in Isabelle

Proof heuristics

- Configure Isabelle simplifier with default rewrite rules
- Design efficient proof tactics for solving common subgoals (continuity and admissibility)

Definition packages

- Implement commands (**fixrec**, **domain**) using *definitional approach*
- Write code for processing user specifications
- Write code for dynamically generating proofs of theorems

Definitional approach

LCF-style kernel provides a sound foundation

How to extend the system while preserving soundness?

Axioms for new constants

Extending HOL with new constants or types requires axioms

Not all recursive specifications are sound!

Sound axiom

foo :: nat $\Rightarrow$ nat

foo n = (if n = 0 then 0 else n + foo (n – 1))

Unsound axiom

bar :: nat $\Rightarrow$ nat

bar n = (if n = 0 then 0 else n + bar (n + 1))

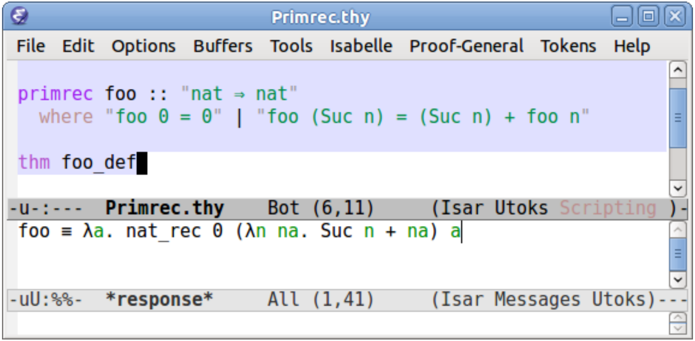
Definitional approach for constants

Non-recursive definition axioms are always safe

- (One definition per constant!)

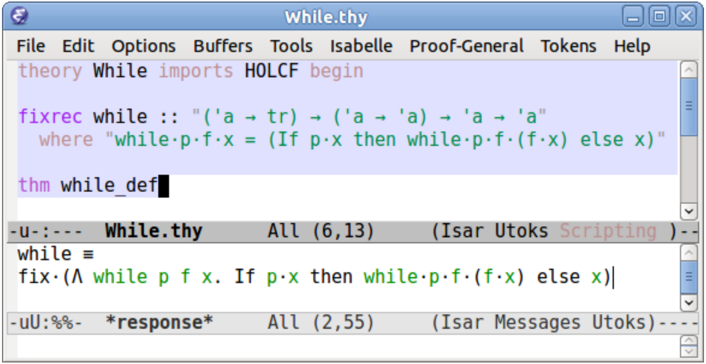
Definition packages

- Turn recursive spec into non-recursive definition
- Derive original recursive equations as theorems



Definition package for HOLCF: Fixrec

- HOLCF-99: users defined recursive functions with “fix”
- HOLCF-11: users can write recursive functions directly



Axioms for new types

Datatype specifications

- list of constructors with argument types
- constructors assumed to be distinct, injective, exhaustive

Not all datatype specifications are sound!

Sound datatype

datatype tree = Tip | Node nat tree tree
(asserts set isomorphism $T \cong 1 + \mathbb{N} \times T \times T$)

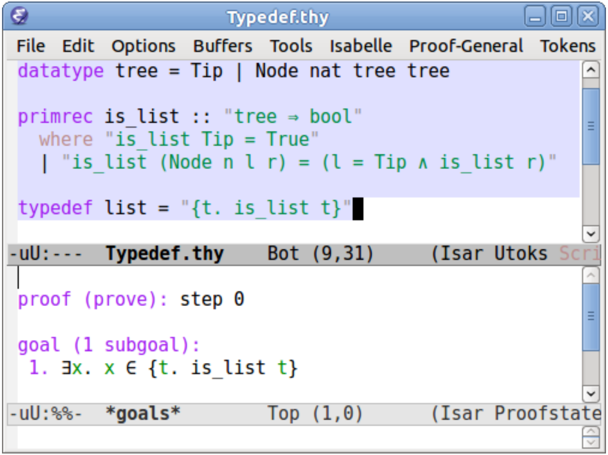
Unsound datatype

datatype object = MkObject nat (object ⇒ object)
(asserts set isomorphism $T \cong \mathbb{N} \times (T \Rightarrow T)$)

Definitional approach for types

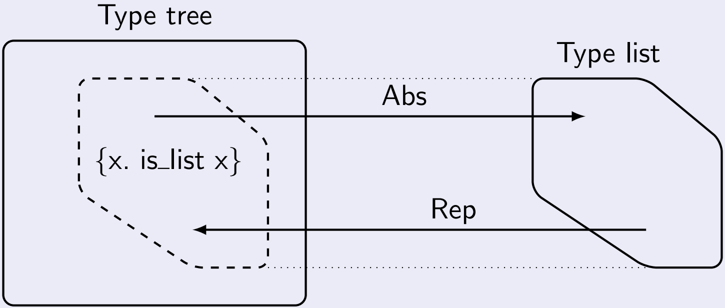
Identify new type with nonempty subset of old type

- Example: Define lists as subset of right-leaning binary trees



Definitional approach for types

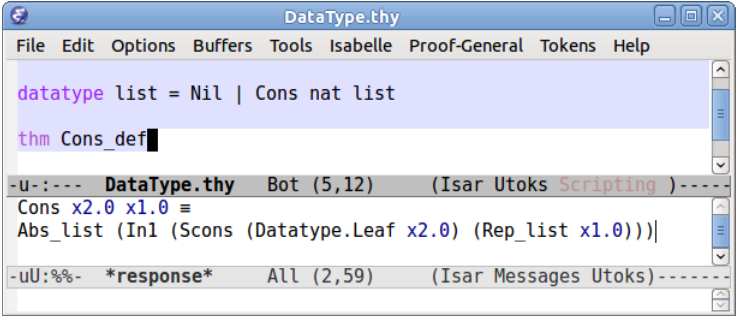
Safe type-definition axioms



$\forall$y. is_list (Rep y)
$\forall$y. Abs (Rep y) = y
$\forall$x. is_list x $\longrightarrow$ Rep (Abs x) = x

Type definition package for Isabelle/HOL: Datatype

- Start with a sufficiently-fancy tree type (Melham 1989, Gunter 1994)
- Define predicate on trees for each new type
- Introduce each new type definitionally
- Define new constructors in terms of Rep, Abs, tree constructors



Type definition package for HOLCF-99: Domain

Analogous to Isabelle/HOL datatype package

- Defines a cpo instead of an ordinary type
- Constructors are continuous functions

Example

domain bintree = Tip | Node bintree bintree
(asserts domain isomorphism $D \cong \mathbb{O} \oplus (D \otimes D)$)

Isomorphism axioms

bintree_abs :: one $\oplus$ (bintree $\otimes$ bintree) $\rightarrow$ bintree
bintree_rep :: bintree $\rightarrow$ one $\oplus$ (bintree $\otimes$ bintree)
$\forall$y. bintree_abs·(bintree_rep·y) = y
$\forall$x. bintree_rep·(bintree_abs·x) = x

Not definitional: Bugs in implementation can make system unsound!

Building a definitional domain package: Universal domain

Definitional approach for types

- Must define new type as subset of existing type
- Problem: What existing type can we use?

Building a definitional domain package: Universal domain

Definitional approach for types

- Must define new type as subset of existing type
- Problem: What existing type can we use?

Solution: *universal domain*

- One big cpo $\mathcal{U}$ with lots of interesting sub-cpos
- Several examples exist in the literature
- Pick a suitable one and formalize it (lots of work!)

Building a definitional domain package: Deflations

Solving domain equations

- Example: $D \cong \mathbb{O} \oplus (D \otimes D)$
- Need type $\mathcal{T}$ whose values identify sub-cpos of $\mathcal{U}$
- $\mathcal{T}$ must be a cpo, to permit recursive definitions
- Type constructors as continuous functions $\mathcal{T} \to \mathcal{T}$

Building a definitional domain package: Deflations

Solving domain equations

- Example: $D \cong \mathbb{O} \oplus (D \otimes D)$
- Need type $\mathcal{T}$ whose values identify sub-cpos of $\mathcal{U}$
- $\mathcal{T}$ must be a cpo, to permit recursive definitions
- Type constructors as continuous functions $\mathcal{T} \to \mathcal{T}$

Solution: *deflations*

- Continuous functions $\mathsf{d} :: \mathcal{U} \to \mathcal{U}$ where $\mathsf{d}(\mathsf{d}(\mathsf{x})) = \mathsf{d}(\mathsf{x}) \sqsubseteq \mathsf{x}$
- Image set is a sub-cpo of $\mathcal{U}$
- Ordered pointwise, deflations form a cpo

Deflation model of types

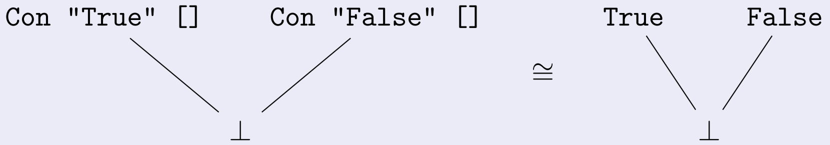
Universal Haskell datatype

```
data U = Con String [U]
```

Deflation for `data Bool = True | False`

```
dBool :: (U -> U)
dBool (Con "True" []) = Con "True" []
dBool (Con "False" []) = Con "False" []
dBool _ = ⊥
```

Image set of `dBool` $\cong$ type `Bool`



Building a definitional domain package

Deflation combinators represent type constructors

Deflation for data `Prod a b = Pair a b`

```
dProd :: (U -> U) -> (U -> U) -> (U -> U)
dProd a b (Con "Pair" [x, y]) = Con "Pair" [a x, b y]
dProd a b _ = ⊥
```

Solving domain equations with deflation combinators

Deflation for domain equation `Stream` $\cong$ `Prod Bool Stream`

```
dStream :: (U -> U)
dStream = dProd dBool dStream
```

HOLCF-11 Definitional Domain package

Steps to define a new datatype

1. Define deflation using deflation combinators and “fix”
2. Define new type as image set of deflation
3. Prove isomorphism “axioms” as theorems
4. Continue as before

No more axioms!

Conclusions

Evaluation

Expressiveness

- Fixrec and Domain packages can handle a wide variety of programs
- Higher-order logic for expressing properties

Automation

- Fixrec and Domain packages generate lots of useful theorems
- Easy subgoals solved automatically by the simplifier

Confidence

- HOLCF-11 is completely definitional
- To trust HOLCF: only need to trust Isabelle's kernel (and axioms of set theory)

More evidence: Concurrency monad case study

Type used to model concurrent computations (Papaspyrou, 2001)

$$\mathbf{R} \cong \mathbf{A} + (\mathbf{S} \to \mathcal{P}^{\natural}(\mathbf{S} \times \mathbf{R}))$$

- Approximated by Haskell datatype:

```
data R s a = Done a | More (s -> [(s, R s a)])
```

- Lots of indirect recursion (functions, pairs, powerdomains)

More evidence: Concurrency monad case study

Type used to model concurrent computations (Papaspyrou, 2001)

$$\mathbf{R} \cong \mathbf{A} + (\mathbf{S} \to \mathcal{P}^{\natural}(\mathbf{S} \times \mathbf{R}))$$

- Approximated by Haskell datatype:

```
data R s a = Done a | More (s -> [(s, R s a)])
```

- Lots of indirect recursion (functions, pairs, powerdomains)

- Proofs about map function, sequencing (functor and monad laws)
  - Papaspyrou's manual proofs: 9–10 pages
  - HOLCF-11: $\approx$ 90 lines
- Hard proof: Associativity of nondeterministic interleaving
  - Manual proof: (none)
  - HOLCF-11: $\approx$ 40 lines

Using HOLCF-11

Anyone can use HOLCF-11

- Part of Isabelle 2011 release
- `http://isabelle.in.tum.de`

Published formalizations (Archive of Formal Proofs)

- Verifying Stream Fusion (Huffman, 2009)
- The Worker/Wrapper Transformation (Gammie, 2009)
- Shivers' Control Flow Analysis (Breitner, 2010)
- `http://afp.sourceforge.net`

The End

Theory libraries

Domain-theoretic concepts added to HOLCF-11:

- Compact (finite) elements (used for admissibility proofs, and more)
- Deflations
- Class of bifinite cpos
- Ideal completion

Significant new libraries:

- Powerdomains (domain-theoretic analog of powersets)
- Universal bifinite domain

Proof heuristics

Continuity prover

- Continuity subgoals are very common (fixed-points, beta-reduction)
- HOLCF-99: exponential-time continuity proofs
- HOLCF-11: new continuity rules give polynomial running time (quadratic number of steps, cubic time)
- Feasible to work with larger programs now

Others:

- New admissibility rules can handle more predicates (using compactness)
- Numerous tweaks to default simplification rules

Contributions to Fixrec Package

Fixrec now supports many new features

- curried functions (i.e. multiple function arguments)
- mutually-recursive functions
- overlapping patterns
- conditional equations
- strictness annotations
- improved induction schemes

A large case study is published on the Archive of Formal Proofs:

- `afp.sourceforge.net/entries/Stream-Fusion.shtml`

Contributions to Domain Package

The reimplemented domain package supports many new features:

- purely definitional (no new axioms!)
- full (and correct!) support for indirect recursion
- support for unpointed lazy argument types
- integration with fixrec package
- improved support for take induction
- better rewrite rules for definedness, comparisons
- improved speed and scalability

Publications

Relevant Publications

- A Purely Definitional Universal Domain (TPHOLs 2009)
- Verifying Stream Fusion (Archive of Formal Proofs, May 2009)
- Reasoning with Powerdomains in Isabelle/HOLCF (TPHOLs 2008)
- Axiomatic Constructor Classes in Isabelle/HOLCF (with John Matthews and Peter White, TPHOLs 2005)

Meetings

- Isabelle Developer's Workshop (Munich, August 2009)
- Informal Isabelle developer's meeting (CMU, March 2009)