A Framework for Embedding Functional Languages in Isabelle

Brian Huffman

Portland State University

August 9, 2007

Primary Goal:

- We want to reason about Haskell programs in a theorem prover
- Especially systems-oriented Haskell programs

Requirements:

- Must support language features used by such programs
  - overloading, constructor classes
  - monadic code with polymorphism
  - monad transformers
- Reasoning must be sound w.r.t. Haskell semantics

Approach:

- Formalize GHC-Core instead of full Haskell

Full Haskell:

- Large language; syntax has dozens of constructors
- Language extensions and changes are frequent

GHC-Core:

- Small language, easier to formalize all of it
- Fully typed, easy to formalize type checking
- Language is stable, rarely changes
- All Haskell language extensions map into Core
- No need to trust compiler front-end

Shallow Embedding:

- Use theorem prover language features to model object language
    - object language terms $\rightsquigarrow$ denotations
    - object language types $\rightsquigarrow$ theorem prover types
    - object language type checking $\rightsquigarrow$ theorem prover type checking
    - object language reduction steps $\rightsquigarrow$ denotational equality
- Used for my earlier attempts to embed Haskell in Isabelle
- Problems:
    - Isabelle's type system is too small!
    - no standard denotational semantics for Haskell

Deep Embedding:

- Model object language features explicitly
  - object language expressions $\rightsquigarrow$ expression syntax
  - object language types $\rightsquigarrow$ type syntax
  - object language type system $\rightsquigarrow$ syntactic typing relation
  - object language reduction steps $\rightsquigarrow$ syntactic reduction relation
- Avoids earlier problems:
  - not limited by Isabelle's type system
  - Haskell has standard operational semantics
- Support for meta-reasoning (induction over syntax)
  - possible to prove parametricity theorems

Denotational Semantics:

- To reason denotationally, use a meaning function:

```
meaning :: exp => univ_domain
ty_meaning :: ty => univ_domain set
```

- Can prove soundness of denotational semantics (with respect to syntactic observational equivalence)
- Can use HOLCF/domain theory, but we aren't tied down to it
- Possibility of using several different meaning functions (use observational equivalence to combine results)

Representing Bound Variables:

- Several possibilities exist for deep embeddings
- Each choice involves trade-offs
  - reasoning about meta-properties
  - reasoning about object programs
  - faithfulness, efficiency of encoding
  - ease of formalization

Explicit Names:

- λf. (λx. g (f x))
  $\leadsto$ Abs 'f' (Abs 'x' (V 'g' • (V 'f' • V 'x')))
- Preserves variable names
- Good for proving meta-properties
- Must reason explicitly about substitution, α-equivalence
- Heavyweight representation using strings

De Bruijn Indices:

- λf. (λx. g (f x))
  $\rightsquigarrow$ Abs (Abs (V 2 $\bullet$ (V 1 $\bullet$ V 0)))
- Identifies α-equivalent terms
- Loses naming information
- Hard to read

Nominal Datatypes:

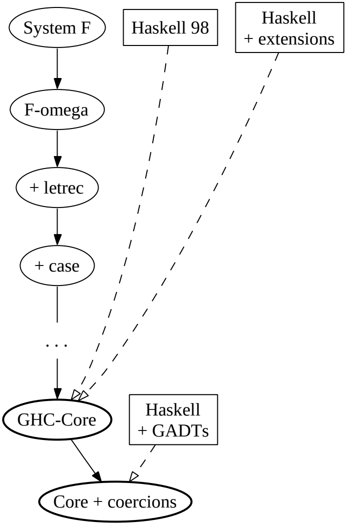
- λf. (λx. g (f x))
  $\rightsquigarrow$ Abs [f]. Abs [x]. V g • (V f • V x)
- Very good for proving meta-properties
- α-equivalent terms are provably equal
- Must still reason explicitly about substitution

Higher Order Abstract Syntax:

- λf. (λx. g (f x))
  $\rightsquigarrow$ Abs (λf. Abs (λx. V 0 $\bullet$ (f $\bullet$ x)))
- Lightweight, good for reasoning about object programs
- Theorem prover takes care of
  - bound variable names
  - α-equivalence
  - substitution
- Support for HOAS not built into Isabelle
  - can define HOAS on top of de Bruijn, with some work

Implementation Roadmap:

- Start with a very basic language (like System F)
- Develop most of the system using the simple language
- Add more language features, one at a time



Datatypes for System F:

```
datatype ty
  = TyVar nat
  | TyFun ty ty
  | TyAll (ty => ty)

datatype exp
  = ExpVar nat
  | ExpApp exp exp
  | ExpAbs ty (exp => exp)
  | ExpTApp exp ty
  | ExpTAbs (ty => exp)
```

This won't work due to higher-order types!

Datatypes for System F: (first-order version)

```
datatype ty
  = TyVar nat
  | TyFun ty ty
  | TyAll' ty

datatype exp
  = ExpVar nat
  | ExpApp exp exp
  | ExpAbs' ty exp
  | ExpTApp exp ty
  | ExpTAbs' exp
```

Operations over types and expressions:

- Lifting (incrementing free variable indices)
- Parallel substitution
- Free variable predicates
- HOAS versions of constructors
- Well-formedness predicates for abstractions

For System F, each comes in 3 flavors:

- types into types
- expressions into expressions
- types into expressions

Problem: Too much boilerplate code

- Lots of work to write (and to read)
- Interesting parts of language definition are obscured
- Object language extensions require lots of new boilerplate
  - Difficult to experiment with small language changes
  - Example: upgrading to new version of GHC-Core

Solution: Automate the process

- A tool can generate constant definitions and proofs

HOAS Datatype package for Isabelle:

```
hoas_datatype ty
  = TyVar nat
  | TyFun ty ty
  | TyAll (ty => ty)

hoas_datatype exp
  = ExpVar nat
  | ExpApp exp exp
  | ExpAbs ty (exp => exp)
  | ExpTApp exp ty
  | ExpTAbs (ty => exp)
```

Give this input to the package; the rest is automatic

Current Status of HOAS Datatype Package:

- Generates inductive datatype definitions
- Generates primitive-recursive definitions for lifting constants
- Supports mutual and indirect recursion

Remaining Work:

- Definitions for substitution, HOAS constructors
- Inductive proofs of lifting/substitution theorems
- Tactics for proving well-formedness of abstractions

Defining Languages using the Package:

- With automatic HOAS definitions, we can focus on the interesting parts of object languages

Typing and Single-step Reduction Relations:

- Inductively defined over syntax
- Additional environment parameters to handle free variables
- Hard to understand with de Bruijn indices

Using HOAS versions of constructors: (Future work)

- Should correspond closely with published language specs
- No need to mention substitution constant explicitly
  - can simply apply function argument of HOAS constructor

Observational equivalence relation: (Future work)

- Defined in terms of single-step reduction relation
- Equivalent terms must have identical termination behavior
- Use greatest such relation that is preserved in all language contexts

Rewriting modulo equivalence: (Future work)

- Possibility: use a set of custom congruence rules for rewriting
- Possibility: take relational image to reduce equivalence to equality
  - $\mathsf{R}\ x\ y \iff (\mathsf{R}\ x = \mathsf{R}\ y)$
- Related work: ACL2 rewriter

Induction over algebraic datatypes: (Future work)

- Very important for usability
- Possibility: Build a language-specific tool to generate induction rules
- Possibility: Simply develop set of lemmas or proof techniques
- Level of automation for end-users should be similar to HOLCF
- Also: induction over recursive function definitions

Evaluating the Framework:

- A proof of separation has been done in HOLCF (by John Matthews)
  - axiomatized monad properties
  - translation by hand into HOLCF
  - fixed monad; not polymorphic
- If successful, the framework should provide a similar level of automation
  - no additional axioms
  - machine-assisted translation
  - full polymorphism

Conclusions:

- Tools and techniques described here will permit formalization of GHC-Core
  - Deep embedding gives flexibility, allows meta-reasoning
  - HOAS facilitates working with object programs
- Much of the framework is general enough to formalize other languages as well
  - Easier incremental development path is possible
  - Maybe part of full Haskell could be formalized as well