Thesis Proposal (Draft)

Brian Huffman

17th December 2007

Abstract

The goal of my thesis project is to create a framework for encoding and reasoning about functional languages in the Isabelle theorem prover. The framework will encode languages using deep embeddings, with higher-order abstract syntax. The motivating test case is the GHC-Core language—an explicitly-typed, desugared variant of Haskell which is used as an intermediate language by the Glasgow Haskell Compiler (GHC).

My thesis statement consists of two parts: First, that the methodology that I present here is a good general way to formalize languages in a theorem prover. In particular, the framework should scale well enough to handle languages at least as complex as GHC-Core. Second, that using a deep embedding with higher-order abstract syntax facilitates doing proofs about object programs. A successful implementation of the framework should provide a level of proof automation comparable to that provided by other libraries (such as Isabelle/HOLCF domain theory).

1 Introduction

Correct computer programs are often very difficult to write. Even when a program appears to work correctly, it may still have problems with rare corner cases—bugs like these can be very difficult to detect, even with extensive testing. In some application areas, an occasional incorrect result or software crash is no big deal, but in some fields, a higher degree of confidence is required. Obvious examples are safety- or security-critical code, where a single software error could be disastrous.

Widely-used library code should also be held to a higher standard, since a single bug can affect a large number of users. Similarly, compile-time optimizations and rewrites also affect large amounts of code: One bug in a compiler can produce bugs in many executable programs, so it is very important to ensure that program transformations performed by a compiler are correct. [A problem with a compiler transformation can affect lots of code! If there are side-conditions on rewrites, these need to be specified precisely. For example, the Haskell foldr/build laws are known not to hold for some programs; only an informal proof of correctness exists.]

One way to obtain a higher level of confidence is to use *formal reasoning*. The term *formal reasoning* refers to proofs that are rigorously constructed and checked by a computer. In contrast, *informal reasoning* includes typical pen-and-paper style proofs and proof sketches, where tedious or uninteresting details are often omitted—these have a much lower level of mathematical rigor.

[TODO: paragraph about Isabelle—Isabelle/HOL is an interactive theorem prover which does formal reasoning.] [TODO: note that in the remainder, "Isabelle" refers to the Isabelle/HOL prover.]

Goals

- Find out how to best reason about modern functional programs using Isabelle: A state-of-the-art theorem prover supporting classical higher order logic.

Non-goals

- Provide a comprehensive framework for reasoning about a specific large language, such as Haskell.

Claims

- Functional language type systems are too difficult to shallowly embed in current theorem provers.
- The best way to proceed is to build a package that can deeply embed the a language's operational semantics for proving meta-properties, and generates HOAS to reason about user program object-properties.
- Can recover most of the convenience of HOLCF/domain theory through language-specific inference rules over the HOAS.
- It is sufficient to first develop these techniques on a compiler's IR. They can then be adapted to the compiler's front-end AST.

What do I mean by "framework", and why is it necessary?

[Proofs about programs and programming languages are often much less interesting than other mathematical proofs. Writing rigorous proofs would be a daunting task, if the theorem prover did not provide some level of automation for common steps. (Not only in constructing proofs, but also for generating definitions.) Automation is important! This is why having a framework for language proofs would be so useful.]

[Formalizations of languages always involve lots of boilerplate, uninteresting definitions, boring proofs... The interesting bits might be hidden away among thousands of lines of proof scripts. By providing automation for the boring parts, a framework can let you focus on the interesting parts. This is obviously better for the person *writing* the formalization—but it is also better for the

person *reading* it, too. By splitting a formalization of a language into a general framework vs. language-specific stuff, it highlights the language-specific parts which are those that you need to trust to ensure that the definitions are right.]

Functional languages are supposed to be easier to reason about than imperative languages. To fulfill that promise, there is a need for more reasoning tools like the ones I present here.

Motivation: What is the problem I am trying to solve?

- Want to prove things about Haskell programs (e.g. safety or functional correctness properties)
- Don't want to have to program in a restricted style (must allow type classes, etc.)
- Must be able to do interactive proof, theorem prover style

Support for reasoning about specific programs demands some additional requirements on the formalization.

Overview of upcoming sections

Section 2 covers the background concepts that readers will need to know about to understand what I am doing, the context of other approaches to solving similar problems, and what design choices I have made.

Section 3 describes the design of the theorem prover framework itself, its components, etc.

Section 4.1 lists some functional languages that the framework should be able to encode, to be used as case studies for the framework.

Section 4.2 describes some object-level programs (in Haskell/GHC-Core) that will be the subject of some correctness proofs.

2 Background

[This section will cover the concepts that readers will need to know about to understand what I am doing, and what design choices I have made.] [In each of these subheadings I will refer to related work: groups and projects that have tried different alternative approaches to mine. (Also approaches that I have previously tried myself!)]

2.1 Lambda calculus

The lambda calculus is a very simple formal language for function definition and function application. A term in the lambda calculus is either a variable such as x; an application of terms t and u, written $(t\,u)$; or an abstraction of a term t over a variable x, written $(\lambda x.\, t)$, which defines a function with x as an input variable. Nested abstractions denote functions that take multiple arguments;

for example, $(\lambda x.\,(\lambda y.\,x))$ represents a function of two arguments, which returns its first argument.

A variable which is enclosed by a matching lambda abstraction is called a *bound variable*; otherwise it is a *free variable*. For example, in the term $(\lambda x.\,f\,(g\,x))$, x is bound, while f and g are free. Note that the names of bound variables are not really important. For instance, the terms $(\lambda x.\,f\,(g\,x))$ and $(\lambda y.\,f\,(g\,y))$ both denote the same function—we say that such terms are *alpha-equivalent*.

Calculation in the lambda calculus is done by *beta-reduction*—this is where an application of a lambda abstraction to an argument is *reduced* by substituting the argument into the body of the function. Formally, we write the beta-reduction rule as $(\lambda x.\,t)\,u \longrightarrow_\beta t[u/x]$, where the notation $t[u/x]$ means to substitute term u for free occurrences of variable x in t. For example, $(\lambda f.\,f\,x)\,(\lambda y.\,y)$ beta-reduces to $(f\,x)[(\lambda y.\,y)/f]$, or $(\lambda y.\,y)\,x$; then this can beta-reduce again, resulting in $y[x/y]$, or x.

Most functional languages (including GHC-Core) are based on the lambda calculus, with some other features added on. One common extension is to add numeric literals and primitive arithmetic operations, so we can write things like $(\lambda x.\,x + 3)$ for a function that adds 3 to its argument. This extension is quite useful, but it also introduces nonsense terms like $5 + (\lambda x.\,x)$ or $(\lambda x.\,3\,x)$, which use functions in place of numbers, or vice versa. This motivates another typical extension: A type system which (among other things) distinguishes functions from numbers. For example, the term 5 has type int, $(\lambda x.\,x + 3)$ has type $\text{int} \to \text{int}$, and $(\lambda f.\,f\,5)$ has type $(\text{int} \to \text{int}) \to \text{int}$.

[Other extensions: datatypes like pairs and tagged unions, along with case expressions * recursive function definitions * recursive datatypes like lists * polymorphic type system: $(\lambda x.\,x)$ may have type $\text{int} \to \text{int}$, $\text{bool} \to \text{bool}$ ... or the polymorphic type $(\forall a.\,a \to a)$ which generalizes all of these. * fancier type systems ... and so on until we get to GHC-Core and beyond.]

2.2 Shallow and deep embeddings

An "embedding" is essentially a translation from one computer language to another: In the context of this paper, the source language is a functional programming language, and the target is the specification language of an interactive theorem prover (Isabelle/HOL). Embedding a programming language into a theorem prover enables formal reasoning about source programs, by proving properties of the translated programs. (Of course, we also need to trust that the translation is correct!)

Specifying a language embedding involves many design decisions and tradeoffs. Perhaps the most important design decision is whether to use a "shallow" or "deep" embedding. A *shallow embedding* takes advantage of common features between the source and target languages, to make the translation as lightweight and simple as possible. With a shallow embedding, a translated source program tends to look like an ordinary program written in the target language. On the other hand, a *deep embedding* basically encodes a source program as a data

structure, which can then be interpreted, analyzed, or manipulated by other programs in the target language.

Shallow embeddings can be very convenient to use, since they can take advantage of the theorem prover's built-in infrastructure. For example, Isabelle represents terms in its logic using a form of simply-typed lambda calculus; notions of alpha-equivalence and beta-reduction are implemented in Isabelle's trusted kernel, so that alpha-equivalent and beta-convertible terms are identified in the logic. If a shallow embedding uses Isabelle's lambda abstraction and function application to represent source-language functions, then alpha-equivalence and beta-reduction come for free—there is no need to manually formalize a reduction relation for the source language.

Another reusable piece of infrastructure is Isabelle's type system: Isabelle has a polymorphic type system with simple type classes and overloading, similar to Haskell. A type checker with automatic type inference is implemented in Isabelle's kernel. If a shallow embedding uses Isabelle's types to represent source-language types, then it can reuse Isabelle's type system and type checker—formalizing the source language's typing rules is unnecessary.

Limitations of shallow embeddings

- target type system might not be fancy enough
- subtle problems caused by similar language features with different semantics
- not possible to do meta-reasoning (i.e. proofs by induction over syntax, quantifying over types...)

[There are limitations to using shallow embeddings. The type system might be too small! There may be subtle problems caused by language features that work slightly differently in the source and target languages (domain package/newtype declarations). (These are the problems I ran into with my first attempt at embedding Haskell in Isabelle, using a shallow embedding.)] [However, it lacks flexibility. We can't just go casually extend Isabelle's trusted kernel to extend the type system or add new language features.]

Benefits/tradeoffs of deep embeddings [A deep embedding models all object language features explicitly. Object-language expressions map onto a syntactic representation in the prover. Object-language types also map onto a syntactic representation.]

Deep embeddings require more work to implement (don't get to reuse prover's built-in type checking, beta-reduction, etc.—require more additional infrastructure). But re-implementing all this stuff does have its advantages. First of all, deep embeddings are more flexible; not using the prover's existing infrastructure means you are not limited by it, and it is possible to specify whatever language features you need. Secondly, defining language concepts explicitly in terms of the source syntax can also bring a higher level of confidence—since such definitions

can often be copied more or less directly from published language specifications. [visual correspondence] [Also, you might be limited by denotational semantics. (What does this mean?) New language extensions are usually specified in the literature syntactically, using operational semantics. By manually formalizing a syntactic reduction relation, it is easier to be sure that you have the definitions right.]

[Meta-reasoning]

How to choose between deep and shallow [In summary, I would say that the choice between shallow and deep depends on the level of sophistication of your source language, how precisely you want to represent that language in the prover, and whether meta-reasoning is important. It is usually better to use a shallow embedding if you can get away with it, but some circumstances demand the power that a deep embedding can provide.]

[Paragraph about previous approaches and related work: Haskell's type system is more sophisticated than Isabelle's, but if you don't need to reason about programs that use fancy type system features, you could still use a shallow embedding. This was our previous approach...]

2.3 Representing bound variables

For deep embeddings, an important design decision is how to represent variables and variable binder syntax. [Mention PoPLMark Challenge.] Several possible approaches exist, and each involves several trade-offs. Factors to consider include: How easy is it to reason about meta-properties, i.e. do pen-and-paper proofs about syntax translate well into the theorem prover? [TODO: define meta-property] Also, how easily does the formalization let one reason about specific object programs? Another important consideration is how much work is involved in formalizing a language. [Also, does the chosen representation faithfully preserve enough information about the original term.]

Why did I choose HOAS? [TODO: maybe move this section earlier?]

Often, a formalization of a programming language is mainly intended for proving meta-theorems about the language itself. Such meta-properties include the subject reduction property or confluence, correctness of type inference, etc. (For example see POPLMark challenge stuff.) In contrast, my formalization is intended to support both kinds of reasoning.

When formalizing a language, an important design consideration is how to deal with variable bindings: Possibilities include de Bruijn indices, explicit names, nominal logic, higher-order abstract syntax, etc. The choice is usually motivated by which system allows easier proofs. For example, proponents of nominal logic claim that it allows proofs that are very close to informal-style paper proofs (when proving meta-theorems, at least).

In contrast to many other formalizations, mine is intended to support two kinds of reasoning: I would like to prove theorems about programs written

in the language, in addition to meta-theorems *about* the language. [A system like nominal logic might be really nice for proving meta-properties, but the encoding for object programs might end up being quite heavy-weight. (For example, consider the statement "variable name x is fresh for term t." This might translate to a very long list of inequalities, if t is a term with a large number of free variables.)] It is my conjecture that a formalization using *higher-order abstract syntax* (HOAS) can scale up well enough to be useful for proving properties of real programs.

Explicit names

The most direct and obvious method of encoding bound variables is to use explicit names. For example, consider the lambda-calculus expression $(\lambda f.\, (\lambda x.\, f\, (f\, x)))$. An encoding of this term in Isabelle might look something like this:

```
Abs "f" (Abs "x" (App (Var "f") (App (Var "f") (Var "x"))))
```

(In the remainder of this paper I will uniformly use math font to represent source language terms, and `typewriter` font for encodings in a theorem prover.)

[Benefits: Easy to define an inductive datatype using Isabelle's datatype package:

```
datatype trm = Var string | App trm trm | Abs string trm
```

Preserves variable names You also need to define an alpha-equivalence relation manually.]

Another drawback is the need to define capture-avoiding substitution. To reason about beta-reduction, you need to manually define a substitution function. With explicit names, you have to worry about name-capture when doing substitution: Sometimes bound variables must be renamed to avoid clashes with free variable names in the substituted term.

[Must reason explicitly about substitution, α-equivalence]

Heavyweight representation using strings (strings are not represented very efficiently in Isabelle—this tends to slow down simplifier, etc.)

De Bruijn indices

The lambda-calculus expression $(\lambda f.\, (\lambda x.\, f\, (f\, x)))$ would be encoded as

```
Abs (Abs (App (Var 1) (App (Var 1) (Var 0))))
```

Identifies α-equivalent terms

Loses naming information

Hard to read

A variable index in a de Bruijn term that is too large for the number of enclosing abstractions is called a *dangling index*. The *proper* de Bruijn terms are those with no dangling indices. For example, in the de Bruijn term `Abs (Var 2)`, the index `2` is dangling, because it is only enclosed by one `Abs` constructor; on the other hand, the term `Abs (Abs (Abs (Var 2)))` is proper. In

some encodings, dangling indices can be used to represent free variables; other encodings might use only proper de Bruijn terms, with dangling indices considered to be ill-formed. [TODO: should I mention locally-nameless style, with de Bruijn indices for bound variables and strings for free variables?]

[Reference for readers unfamiliar with de Bruijn indices: Wikipedia article "de Bruijn index" (this also has some nice graphics) Note that my indices start at zero, while in the examples there indices start counting at 1.]

Higher-order abstract syntax

[TODO: introduce Isabelle lambda-notation here (% is the lambda-binder in Isabelle)]

```
Abs (%f. Abs (%x. App f (App f 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.

The `Abs` constructor takes a function from terms to terms as its argument, returning a new term. In generic theorem prover like Isabelle, it is possible to create functions of the appropriate type that don't correspond to valid terms—consider `Abs (%x. if x = t1 then t2 else t3)`. Such terms are known as *exotic terms*. Some provers (like LF, Twelf) disallow the construction of exotic terms by means of a more restrictive logic. In these provers HOAS can be used freely, but due to the restrictive logic, induction cannot be used freely. In Isabelle, induction can be used without restrictions, but it is necessary to define a predicate to recognize valid functions.

2.4 Operational and denotational semantics

When working with an embedding of a source language, we are interested in reasoning about the evaluation of source language terms. With a deep embedding, there are two main approaches for how to do this: operational or denotational. Operational semantics is done by defining a syntactic reduction relation. With this approach, if you want to show that term `t1` evaluates to term `t2`, you must show `reduces t1 t2`, where `reduces` is a binary predicate that encodes the reduction relation. On the other hand, denotational semantics takes the form of a semantic meaning function: This takes a syntactic term as input, and returns a value from an appropriate value domain. With this approach, instead of showing that two terms `t1` and `t2` are related by some syntactic relation, you show that they have the same meaning, i.e. `meaning t1 = meaning t2`.

If a suitable meaning function can be found, denotational semantics can work very well for reasoning about source programs. A major benefit is that with denotational semantics you get to use equality-based reasoning, which is well-supported by theorem provers like Isabelle: Isabelle's simplifier is designed

Table 1: foo

Denotational model	Limitation
Projections	No forall-types
Complete PERs	No negative recursion
Uniform PERs	Forall-types contain junk

specifically for equality-based rewriting on terms. Additionally, if the value domain is a good fit for the language, you may have the option of proof by induction over the value domain. (For example, in a HOLCF domain-theoretic denotational semantics, we can take advantage of the cpo structure of the value domain and do proofs using least-fixed-point induction.)

The main obstacle to using denotational semantics is that finding a suitable value domain can be very tricky. [TODO: maybe mention how long it took/how much high-powered mathematical machinerywas required to develop denotational semantics for untyped lambda calculus] Furthermore, if you have a denotational semantics for your language, and then extend your language with a small new feature, the old value domain might no longer work—and a new one might be very difficult to find. Also, you may have separate denotational models, each of which supports a different language extension, but not be able to combine them to create one model that works for all language extensions together.

[A good portion of my graduate research so far was spent looking for a suitable denotational semantics for GHC-Core. I evaluated several candidate approaches; each approach worked well on a certain subset of language features that I wanted, but none was able to handle all the features of GHC-Core. A summary of these approaches is shown in Table [].

[Benefit of operational: straighforward translation from published language semantics into a reduction relation. (New languages are usually specified using operational semantics; denotational models come later, if at all.)]

[Benefit of operational: language extensions are usually pretty easy to handle, just add more introduction rules to the reduction relation.]

[Drawback of operational: no equality-based rewriting. We can come up with some work-arounds, but they are more work to implement.]

- list various denotational models, with language features each doesn't handle
- relating the two: showing that one semantics is sound w.r.t. another

[Summary: denotational semantics is nice to use for reasoning about source programs, if you have a denotational model that is a good fit for the language you want to reason about. (I think that operational semantics is well-suited for meta-reasoning, though.) But if you want to play around with language extensions, you will probably need to use operational semantics.]

3 Implementation

"The methodology that I present here is a good general way to formalize languages in a theorem prover using higher-order abstract syntax."

[The framework/methodology is made up of several parts/steps. Some will be tool support (i.e. definition packages implemented as part of Isabelle), some will be libraries of reusable constants and theorems, and some will just be simply a style/pattern of writing definitions/proofs. The framework will be designed so that you can focus on the interesting bits of a language, and the boring redundant bits are more automated.]

[Some parts are already implemented; many other parts I have experimented with; other parts I have thought about. In the unimplemented parts, I will try to identify specific issues that I will have to deal with, and offer some possible design decisions to deal with them.]

3.1 Defining languages with higher-order abstract syntax

The first step in the implementation is to decide how to define higher-order abstract syntax: This includes datatype declarations, definitions of substitution and other related functions, and finally the HOAS constructors themselves.

Encoding with de Bruijn indices

A simple version of the untyped lambda-calculus will serve as a running example. A term in this language is either a free variable v_n (indexed by a natural number n), an application of two terms $t \bullet u$, or a lambda abstraction $(\lambda x.\ t)$ (where x is a variable name that may occur in t). Term application associates to the left, so that $f \bullet x \bullet y$ means the same as $(f \bullet x) \bullet y$.

A first attempt to encode this language in Isabelle might look like this:

```
datatype trm = Var nat | App trm trm | Lam (trm => trm)
```

However, this datatype declaration is not allowed, since it has a negative occurrence of the defined type (shown underlined). An injection from the entire function space `(trm => trm)` into `trm` would cause a logical inconsistency, and datatype constructors are always injective, so Isabelle's datatype package is forced to reject this definition.

Instead, we replace the higher-order constructor `Lam` with `Abs`, which takes a single term argument and relies on de Bruijn indices to represent bound variables. The new type definition is shown in Figure 1. In this encoding, the constructor `Var` does double duty, representing both bound and free variables. For example, `Abs (Abs (App (Var 3) (App (Var 1) (Var 0))))` represents the term $(\lambda f.\ (\lambda x.\ v_1 \bullet (f \bullet x)))$. In a context of two enclosing `Abs` constructors, `Var n` represents a bound variable if `n < 2`; otherwise, `Var n` represents the free variable with index `n - 2`.

```
datatype trm = Var nat | App trm trm | Abs trm
type env = (nat => trm)

lift :: trm => nat => trm
lift (Var n) i = Var (if n < i then n else Suc n)
lift (App t u) i = App (lift t i) (lift u i)
lift (Abs t) i = Abs (lift t (Suc i))

envlift :: env => env
envlift σ 0 = Var 0
envlift σ (Suc n) = lift (σ n) 0

subst :: trm => env => trm
subst (Var n) σ = σ n
subst (App t u) σ = App (subst t σ) (subst u σ)
subst (Abs t) σ = Abs (subst t (envlift σ))
```

Figure 1: Definition of terms, parallel substitution and related functions

Lifting and substitution

Once we have a datatype for encoding our language, the next step is to define a "lifting" operation, which will be needed in order to define substitution. The expression `lift t n` increments by one all free variables in term `t` whose indices are greater than or equal to `n`. In particular, in the common case where `n = 0`, the expression `lift t 0` adds one to the index of *all* free variables. Lifting allows you to enclose a term `t` under an additional `Abs` constructor, while preserving the meaning of its free variables.

Let us define a "term environment" to mean a function from free variable indices (i.e. natural numbers) to terms. In addition to the lifting operation `lift` for terms, we will also need a lifting operation `envlift` for term environments. When performing substitution under an `Abs` constructor, it is necessary to modify the term environment so that the free variable indices are consistent. The `envlift` function does exactly what is needed; its definition is shown in Figure 1.

Finally, we can use `envlift` to define a parallel substitution operator `subst`, which replaces the free variables in a term with other terms. It is called "parallel" because it replaces all free variables simultaneously. More specifically, `subst t σ` replaces all references to free variable `n` in `t` with `σ(n)` (appropriately lifted so that the free variable indices still make sense). Bound variables are left untouched.

The definitions of lifting and substitution may be easier to understand after working through an example. Define term `t` as `App (Abs (App (App (Var 2) (Var 0)) (Var 1))) (Var 0)`, which is the encoding of the source term $(\lambda x.\, v_1 \bullet x \bullet v_0) \bullet v_0$. Also let term environment σ be defined such that `σ(0) = Var`

`4` and σ`(1) = Abs (Var 0)`, which encode v_4 and $(\lambda y.\, y)$ respectively. Then by unfolding the function definitions, the reader can verify that `subst t` σ returns `App (Abs (App (App (Abs (Var 0)) (Var 0)) (Var 5))) (Var 4)`. This is in fact the encoding of the source language term $(\lambda x.\, (\lambda y.\, y) \bullet x \bullet v_4) \bullet v_4$—which is simply $(\lambda x.\, v_1 \bullet x \bullet v_0) \bullet v_0$ with v_4 substituted for v_0 and $(\lambda y.\, y)$ for v_1.

Single-variable substitution

Any formalization of substitution needs to be able to describe beta-reduction, a common operational step in functional programming languages. In our de Bruijn representation, a beta-reducible expression (redex) has the form `App (Abs t) u` for some `t` and `u`. We obtain the result of the reduction by replacing free occurrences of `Var 0` in `t` with `u`—and since we are removing an `Abs`, we also need to decrement the indices of all other free variables in `t`. It is straightforward to express this single-variable substitution/decrement in terms of parallel substitution:

```
subst1 t u = subst t (%n. if n=0 then u else Var (n-1))
```

Other formalizations of substitution [Nipkow, "More Church-Rosser Proofs"; ACM02 Hybrid paper] define single-variable substitution directly, without reference to parallel substitution. I have tried formalizing several variations on substitution, and I settled on the current parallel substitution approach for several reasons:

- Single-variable substitution is easy to define in terms of parallel substitution, but not vice-versa.
- When single-variable substitution also decrements indices, many lemmas are quite complicated [Nipkow; Urban?, comparison paper], involving side-conditions and "`+1`" on indices in many places. Parallel substitution satisfies nicer properties which are much simpler to state, and also easier to prove.
- Parallel substitution generalizes well to languages having multiple sorts of binders, like System F_2. (I will have say more about this in the next section. [TODO: cross-ref])

HOAS constructors

The function `subst1` defined above is a binary operator on terms, i.e. it has type `trm => trm => trm`. We can also think of it as having type `trm => (trm => trm)`, taking a single term argument and returning a function. For example, `subst1 (App (Var 0) (Var 2))` evaluates to the function `(%u. App u (Var 1))`.

A HOAS constructor like `Lam` should basically do the opposite of `subst1`: Instead of taking a term and returning a function, it takes a function and returns a term. For example, when we apply `Lam` to the function `(%u. App u (Var 1))`,

```
envlift (%n. Var n) = (%n. Var n)
subst t (%n. Var n) = t

lift (lift t i) 0 = lift (lift t 0) (Suc i)
envlift (%n. lift (σ n) i) =
     (%n. lift (envlift σ n) (Suc i))

lift (subst t σ) i = subst t (%n. lift (σ n) i)
envlift (%n. subst (σ n) σ') =
     (%n. subst (envlift σ n) (envlift σ'))
subst (subst t σ) σ' = subst t (%n. subst (σ n) σ')
```

Figure 2: A selection of lemmas about lifting and parallel substitution

we should get the term `Abs (App (Var 0) (Var 2))` (the original term above, but with an `Abs` constructor in front). In general, `Lam` must satisfy the equation `Lam (subst1 t) = Abs t`, for all terms `t`.

There are a few possible alternatives for defining `Lam`. One possibility is to use Isabelle's definite choice operator: Define `Lam f` to be `Abs t`, for the unique `t` such that `subst1 t = f`. This makes for a nice, short definition, but it also requires a (possibly tricky) uniqueness proof to ensure that the term `t` is well-defined.

It is possible to avoid the well-definedness proof obligation, at the expense of a slightly more complicated definition. [This approach is from Capretta/Felty '06] For example, if `Lam` is given a function like `(%u. App u (Var 1))` as an argument, it can apply that function to `Var k` resulting in `App (Var k) (Var 1)`. (Let us ignore for the moment where `k` comes from.) Next it can apply a substitution that maps `k` to `Var 0` and increments all other variable indices, giving `App (Var 0) (Var 2)`. (Note that `k` must be distinct from `1` for this to work!) Finally it can put an `Abs` constructor on the front to give the desired result.

Of course, we also need some way to choose `k` to be distinct from all other free variable indices. This could be accomplished by defining a function `newvar` that calculates an unused variable index for a given term [Capretta/Felty 2006, "Combining de Bruijn Indices and HOAS in Coq"]. Alternatively, since we get the same result for all but finitely many choices of `k`, it is also possible to use a kind of choice operator to choose a "big enough" value of `k` automatically.

[I have tried both kinds of definitions in my manual developments.] The final choice of which style to use [for the more complex languages] will be based on evaluating trade-offs between simplicity of definitions versus ease of constructing proofs.

3.2 Generalizing and automating HOAS definitions

The previous section showed how to define an encoding with higher-order abstract syntax for a very simple functional language. I would like to be able to use similar techniques to encode significantly more complex languages, like GHC-Core. However, instead of jumping right in with the full GHC-Core language, I plan to proceed in an incremental fashion: Starting with a stripped-down version of GHC-Core, I will develop a language encoding along with its typing rules, operational semantics and so on. Then I plan to add more language features one at a time, updating all the definitions and proofs as necessary.

Generalizing to System F_2

The untyped lambda-calculus language from the previous section has only one sort of entity: everything is a term. Now we will extend the language by adding another sort of entity—types. In the language System F_2 (also known in the literature as simply "System F"), a type τ is either a free type variable t_n (indexed by a natural number n), a function type $\tau \to \tau'$, or a universal type $(\forall \alpha.\, \tau)$ (where α is a type variable name that may occur in type τ). Function arrows associate to the right, so that $\tau \to \tau' \to \tau''$ means the same as $\tau \to (\tau' \to \tau'')$.

Terms in System F_2 are built similarly to terms in the untyped lambda-calculus, but with a few additions. First of all, every lambda abstraction in System F_2 is annotated with an argument type τ, as in $(\lambda x^{\tau}.\, t)$. Also, we add two new term constructions: abstraction of a term t over a type variable name α, written $(\Lambda \alpha.\, t)$, and application of a term t to a type τ, written $t \diamond \tau$.

If we could encode System F_2 directly with higher-order abstract syntax, the datatype declarations might look something like this:

```
datatype ty = TVar nat | Fun ty ty | Forall (ty => ty)

datatype trm = Var nat | App trm trm | Abs ty (trm => trm)
             | AppT trm ty | AbsT (ty => trm)
```

As before, this declaration will not work as written because of negative recursion (shown underlined). The actual datatype declarations will have the underlined parts removed, again relying on de Bruijn indices to encode bound variables.

The `Var` constructor is interpreted the same as with the untyped lambda calculus: `Var n` refers to a bound variable if it is surrounded by more than `n Abs` constructors; otherwise it refers to a free term variable. The interpretation of `TVar` is a bit more interesting, since there are two different binders for types in System F_2: Depending on context, `TVar n` may refer to a variable bound by an enclosing `Forall` constructor, a variable bound by an `AbsT` constructor in an enclosing term, or a free type variable. For example, `AbsT (Abs (Forall (Fun (TVar 0) (TVar 1))) (Var 0))` encodes the System F_2 term $(\Lambda \alpha.\, (\lambda x^{\forall \beta.\, \beta \to \alpha}.\, x))$, where the type variable β is bound with a `Forall`, and α is bound with an `AbsT`.

```
type tyenv = (nat => ty)
type trmenv = (nat => trm)

ty_lift :: ty => nat => ty
trm_lift :: trm => nat => trm
trm_ty_lift :: trm => nat => trm

ty_envlift :: tyenv => tyenv
trm_envlift :: trmenv => trmenv
trm_ty_envlift :: trmenv => trmenv

ty_subst :: ty => tyenv => ty
trm_subst :: trm => trmenv => tyenv => trm
```

Figure 3: Lifting and substitution operators for System F_2

In order to define HOAS constructors for this encoding of System F_2, we start by defining lifting operators, just like with the untyped lambda-calculus. But because of the multi-sorted encoding of System F_2, with `ty` separate from `trm`, we will need multiple lifting operators: `ty_lift` increments the free type variables in types, `trm_lift` increments the free term variables in terms, and `trm_ty_lift` increments the free type variables in terms. Similarly, we also need three `envlift` functions: one for type environments and two for term environments. (To go with these, we also need three variants of each lemma about lifting from Figure 2.)

Fortunately, using parallel substitution allows us to get away with having only two substitution operators, one for types and one for terms. The term substitution operator `trm_subst` takes both a term environment and a type environment as arguments, and performs substitution on both sorts of variables simultaneously. With single-variable substitution, three different versions would be needed, along with a family of lemmas to show how they interact with each other. I have found that parallel substitution is a big win for formalizing System F_2, and I expect it to scale much better than single-variable substitution to larger multi-sorted languages.

[TODO: reference to Figure 3]

HOAS datatype package for Isabelle

A problem with defining higher-order abstract syntax manually is that it is hard to experiment with language extensions. The manual formalization of untyped lambda-calculus was quite reasonable, requiring 4 or 5 constant definitions and about 20–30 lemmas to define an encoding with HOAS. [TODO: refer to Figure 2] I have also done a manual formalization of System F_2, which involves around a dozen constant definitions and more than 60 lemmas. Beyond this point, manually updating the definitions to incorporate new language features begins

to be a problem. To add a single new constructor to the language, several dozen definitions and proof scripts need to be modified.

The latest version of GHC-Core [SCJ06] is based on a language called System F_C, which is multi-sorted to a greater degree than System F_2. In addition to terms and types, System F_C also includes two new sorts of entities: kinds and type coercions. Furthermore, System F_C involves plenty of mutual recursion: Types, kinds, and coercions can all refer to each other, and as a consequence, more than a dozen varieties each of `lift` and `envlift` are necessary. Without some form of automation, a formalization of HOAS for such a complex language would be tedious to write and difficult to maintain—for this reason I have not attempted a manual formalization of System F_C.

In the formalization of System F_2, most of the numerous definitions and proof scripts are repetitive and uninteresting—the definitions and proofs for System F_2 follow the same basic patterns as those for the untyped lambda calculus. Long, repetitive, boring definitions and proofs are ill-suited to writing by hand, but these are exactly the qualities that automatic tools excel at.

[What does the tool do? * Define datatype * Define substitution/lifting functions, free variable predicates * Prove standard lemmas about these * Define HOAS constructors]

[TODO: mention how much is already finished with the package]

[Related work: nominal-datatype package, Hybrid, Poplmark challenge]

3.3 Defining typing relation and operational semantics

Let's say we are defining a typing relation for System F_2 called `has_typ`, which has type `trm => typ => bool`. To show that a lambda abstraction Lam t1 f has a the function type Arrow t1 t2, We might like to have something like this as the typing rule for abstractions:

```
(ALL x. has_typ x t1 ==> has_typ (f x) t2)
  ==> has_typ (Lam t1 f) (Arrow t1 t2)
```

[These would be defined as inductive relations, in terms of HOAS syntax. Ideally the definitions would look very similar to how these things would be written on paper. We could just use Isabelle's built-in inductive definition package, probably no special tool support necessary.]

[My earlier attempts: Defining typing relation, etc. directly over de Bruijn terms. The problem is that the definitions are decorated with variable-reindexing and lifting functions, and are hard to read and understand. This is a problem, because we have to trust the correctness of these definitions; it is important that their correctness be verifiable by visual inspection.]

This rule, with its nested implication, is an example of "hypothetical judgments". In theorem provers like Twelf, this can be used [Pientka paper] but in Isabelle and Coq, this is an instance of non-positive recursion and is rejected by the logic. (At least in the case where we are using the same predicate we are defining—has_type—on the left side of the implication. This actually would work with weak HOAS.)

Instead, I plan to formalize the typing relation using environments. For example, in System F2 I will define a predicate has_type :: typenv => trm => typ => bool. The rule for abstractions will be something like:

(EX k. fresh k f /\ has_type (env (k:=t1)) (f (Var k)) t2)

==> has_type env (Lam t1 f) (Arrow t1 t2)

One interesting issue to consider is how to handle the choice of a new fresh variable in inductive rules having to do with binding syntax. (Show example typing rules for the lambda-calculus language above. Point out free variable side-condition in abstraction rule.) By the way, this issue is not specific to HOAS; it comes up in nominal logic and other variable codings as well. Different quantifiers are possible (existential, universal, cofinite), and the choice affects inductive proofs of meta-theorems about typing. [Related work: Hybrid, Randy Pollack and related papers discuss definition of relations like these over HOAS.]

3.4 Defining operational equivalence relation

[Operational equivalence is defined in terms of the small-step operational semantics. There would also be termination predicates, etc. We'll have to wait and see how much of this step can be generalized and reused between languages; probably no tools necessary, but maybe some convenient functions could be made to assist in the definitions.]

[There will be a standard set of proofs about operational equivalence: Term constructors respect equivalence, etc.]

[TODO: reference to Pitts 2000 paper]

Rewriting modulo equivalence relations

[We want to be able to treat operational reduction steps like rewrite rules. The simplifier will have to be configured somehow to allow this. Design decision: Maybe we could define a quotient type, so that contextual equivalence is just equality?]

3.5 Proofs by cases and induction

Many interesting proofs rely on case analysis and induction over datatypes, such as the list datatype. For example, mapping the identity function over a list gives back the original list (map id xs = xs); to prove this, some form of induction is necessary. [In Isabelle/HOL, we can prove "map id xs = xs" by induction on xs, proving a base case for the empty list, and proving an inductive case for the cons constructor. In Isabelle/HOLCF, where lazy lists may be infinitely long, the induction rule has an additional admissibility requirement, which checks that the predicate (%xs. map id xs = xs) holds for an infinite list when it holds for all of its finite approximations.]

[Another type of inductive proof uses induction over recursive function definitions. For example, instead of proving "map id xs = xs" by induction over the

list datatype, another way to proceed is to use induction over the definition of map.]

For my framework to be useful for proving properties of real programs, it is necessary that inductive proofs be possible.

[It would not be worthwhile for me to create packages to fully automate this. Any automated tools that I could create would be language-specific for GHC-Core, so there is much less potential for reuse.]

[There needs to be a way to derive induction rules for recursively-defined datatypes in the object language. At the least, this will be a general proof strategy, exemplified by hand-made induction rules for a few fixed datatypes (like lists). At best, this could be tool support for automatically generating and proving induction rules from type definitions.] [This stuff will probably be fairly object-language-specific, any tools that I make will probably be just for GHC-Core.]

[Same comments as previous section. We must also consider what side conditions the induction rules will have: Will there be some sort of admissibility requirement? Will the proofs look more like induction or coinduction?]

4 Evaluation Criteria

[What do I need to do to demonstrate the success of my approach?]

4.1 Language Case Studies

Tool support means that it should be easy to define new languages.

* Poplmark challenge language: F-omega with subtyping [easy example]

* GHC-Core [this will be the real stress-test for the framework]

* Multi-level security information-flow type system [good security-related example]

* Subset of surface-level Haskell [proof of concept for future work?]

[What does "easy to define" mean? First of all, the essential definitions of an object language (i.e. typing/evaluation relations) should be straightforward to define, easy to understand, and obviously correspond with written standards.]

4.2 Proofs on Object Programs

"Using a deep embedding with higher-order abstract syntax facilitates doing proofs about object programs, in addition to proofs of meta-properties."

To establish the second part of my thesis statement, it will be necessary to do some proofs about some object programs. (Choose one or more of the following)

- Simple proof of separation for 2-process round-robin scheduler: [This would basically look like John's HOLCF proof. If all goes as planned the proof would have a similar degree of automation.]

- Prove that a given monad implementation, show that it satisfies a certain set of properties (such as those assumed for scheduler example)
- Direct comparison with Nominal-datatype package (do a similar example)

5 Timeline

...

References

[SCJ06] Martin Sulzmann, Manuel Chakravarty, Simon Peyton Jones. System F with Type Equality Coercions. 2006.

[AM00] Andrew W. Appell, David McAllester. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. 2000.

[Ahmed06] Amal Ahmed. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. 2006.

[ACM02] S. J. Ambler, R. L. Crole, A. Momigliano. Combining Higher Order Abstract Syntax with Tactical Theorem Proving and (Co)Induction. 2002.

[CF06] Vananzio Capretta, Amy P. Felty. Combining de Bruijn Indices and Higher-Order Abstract Syntax in Coq. 2006.

[Nip99] Tobias Nipkow. More Church-Rosser Proofs (in Isabelle/HOL). Journal of Automated Reasoning, 21:51–66, 2001.

[Huet94] Gerard P. Huet. Residual theory in lambda-calculus: a formal development. Journal of Functional Programming, 4(3):371–394, 1994.

[Alt92] Thorsten Altenkirch. A Formalization of the Strong Normalization Proof for System F in LEGO.