Proposal: SLYCE

Raven Rothkopf (rgr2124) and Gregory Schare (gs3072)

Introduction

Our proposal is to write an interpreter in Haskell that implements a toy language, called SLYCE[1], featuring dependent types. Our language will look roughly like an ML-style language; it will feature lambda abstractions, let-expressions, if-then-else expressions, and some typical primitives like basic Boolean and arithmetic operators. It will be pure. It will have `Boolean`, `Integer`, `Char`, and `String` types built-in[2], and it will allow the user to build types parameterized by terms and other types, such as `Pair T1 T2`, which describes a pair of values where the first has type `T1` and the second has type `T2`; and `List n T`, which describes a finite list of `n` values each of type `T`.

Syntax and Semantics

A grammar for our language might look something like this:

```
e ::= x                   Variable
   |  lambda (x…) e       Abstraction
   |  e e                 Application
   |  let x = e in e       Let-Binding
   |  if e then e else e   Conditional
```

Notably, we are missing the type annotations. We are sure that we must include some types in the program (in order to facilitate bidirectional typing) but as of right now we are not sure which terms should be annotated by types.

Every SLYCE program is an expression. The value of a SLYCE program is the evaluation of this expression, and this is what gets printed to the console when you interpret the program.

We want SLYCE to be strongly normalizing, i.e. every well-typed term reduces to some value. This means it will not be as powerful as the untyped lambda calculus, but we gain a guarantee of consistency.

---

[1] SLYCE is a reference to the phrase "slice of pie", because pie sounds like Π, as in the Π-calculus, which is the language of dependent types. However, since our project is a simple toy implementation, it is a "slice of Π". Moreover, the "y" in the name looks like an upside-down λ, evoking how our language differs from the λ-calculus.

[2] In the event that our project scope is too large, this is an obvious place to cut down. We can get away with just Boolean and natural number types for the basic types and just `List` for the types parameterized by other types.

Dependent Types

The key feature we would like to explore is *dependent types*. We are still in the process of learning dependent types, but one helpful way to understand it is from the Barendregt's lambda cube:

- terms that bind types correspond to polymorphism
- types that bind types correspond to type constructors (as we saw in Hindley-Milner-Damas)
- types that bind terms correspond to dependent types

(Obviously, in untyped lambda calculus and beyond, terms can bind other terms via lambda abstraction.)

In other words, types in SLYCE can be parameterized both by terms and by other types. Dependent types break down the distinction between types and terms. This makes it all the more important to be aware of the differences.

Example: type signatures featuring lists of a particular length

The "hello world" of dependent types is the ability to type-check the canonical `map` function on lists and prove that it outputs a list of the same length as its input list:

```
map :: [A:Type] -> [B:Type] -> [n:Nat] -> (A -> B) -> List A n -> List B n
```

Note how this differs from refinement types, in which one can express complex relationships between the values of inputs and outputs.

```
map :: (a -> b) -> (xs : list a) -> (ys : list b) where len xs = lens ys
```

There is some desiderata here from refinement types: in order to express more complex relationships, we want the ability to perform simple operations on the terms in our type signatures. In Stephanie Weirich's tutorial, and in the dependently-typed language Pie, one can only express relationships between terms in a type signature using type constructors, such as `Succ` for natural numbers.

For example, the `cons` function proves that it outputs a list that increases the length of the input list by 1:

```
cons :: [A:Type] -> [B:Type] -> [n:Nat] -> A -> List A n -> List A (Succ n)
```

Similarly, `filter` always returns a list that is at most smaller than its input list. It is easy to write this with refinement types:

```
filter :: (a -> Bool) -> (xs : list n a) -> (ys : list m a) where m <= n
```

We are not sure if it is possible to express this with dependent types. Perhaps one can use some sort of type scheme, wherein the output list `ys` can be any concrete type of a type scheme describing lists of `A` whose length is any natural number `m` less than or equal to `n`. We are not sure if it is possible to combine dependent types and polymorphism in this manner.

`zipWith` is another interesting case. There are two ways to define `zipWith`. In the first (and simpler) case, we require that both input lists have the same length and we output a list of that length.

```
zipWith :: [A:Type] -> [B:Type] -> [C:Type] -> [n:Nat] -> (A -> B -> C) ->
(List A n) -> (List B n) -> (List C n)
```

But if we were able to write more complex relationships between the types, we could write a `zipWith` type signature in which the output list has the length of whichever input list is smaller:

```
zipWith :: (a -> b -> c) -> (xs : list a n) -> (ys : list b m) -> list (min2 n m) c
```

We would like to explore these possibilities.

Implementation

We have amassed a number of resources for learning the theory of dependent types and their implementation, and we have just begun to dive into them. Some of these resources include Stephanie Weirich's tutorial, Appendix B of *The Little Typer*, Löh and McBride's 2009 tutorial, and Nikita Voloboev's personal wiki containing a bunch of links to other resources.

We are aware that one of the central questions with dependent types is how the system can type-check types that depend on run-time values; it is clear that some compile-time evaluation is necessary. This amounts to the question of *normalization.* This is why it is critical that our system is strongly normalizing. One system for dependent types that appears to satisfy this is the calculus of constructions (used in Coq). We found a tutorial describing implementation of the CoC in OCaml, and we hope this will serve as a useful guide for our implementation. However, we are currently still in the process of understanding CoC.

As John mentioned in the EdStem post, we will likely explore the use of bidirectional type checking, which Stephanie Weirich uses in her tutorial.

If we have time, we hope to also include a simple scanner and parser so that we can write SLYCE programs outside of the Haskell AST.

General Questions

How is the Calculus of Constructions more expressive than lambda calculus?
As we mentioned above, we have found resources that implement dependent types for the Calculus of Constructions. We are aware that CoC is more expressive than the λ-calculus, but we aren't entirely sure why.

How does bidirectional typing help us implement dependent types?

What is the relationship between refinement types and dependent types?

Which terms in our grammar should be annotated by types?