Types for binding

Paul Stansifer

April 13, 2012

Let's consider an even more simply-typed λ-calculus. You might recognize this as the Bob Harper-ly typed λ-calculus.

$$\begin{array}{rcl} e & ::= & x \mid \lambda x{:}\tau.e \mid e\ e \\ \tau & ::= & \mathbf{ok} \\ \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \end{array}$$

I bet you can figure out the type rules:

$$\frac{\Gamma(x) = \mathbf{ok}}{\Gamma \vdash x : \mathbf{ok}}\ \textsc{T-Relax} \qquad \frac{\Gamma, x{:}\tau \vdash e : \mathbf{ok}}{\Gamma \vdash \lambda x{:}\tau.e : \mathbf{ok}}\ \textsc{T-ChillOut}$$

$$\frac{\Gamma \vdash e_0 : \mathbf{ok} \qquad \Gamma \vdash e_1 : \mathbf{ok}}{\Gamma \vdash e_0\ e_1 : \mathbf{ok}}\ \textsc{T-Whatever}$$

Well, it does check for unbound names, at least. That's not nothing. But it's pretty easy. Why don't we make it a goal to do something like this, but harder?

But first, let's move towards a more practical type system.

$$\begin{array}{rcl} e & ::= & x \mid \lambda x{:}\tau.e \mid e\ e \\ \tau & ::= & \mathbf{ok} \mid \tau \to \tau \\ \Gamma & ::= & \cdot \mid \Gamma, x : \tau \end{array}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \textsc{T-CoolVar} \qquad \frac{\Gamma, x{:}\tau_0 \vdash e : \tau_1}{\Gamma \vdash \lambda x{:}\tau_0.e : \tau_0 \to \tau_1}\ \textsc{T-Lambda'sGood}$$

$$\frac{\Gamma \vdash e_0 : \tau_0 \to \tau_1 \qquad \Gamma \vdash e_1 : \tau_0}{\Gamma \vdash e_0\ e_1 : \tau_1}\ \textsc{T-GoAheadAndApply}$$

This is like the STλC, but without base types. We still haven't got any interesting stuck states, but we're now requiring that the programmer express how deeply they're going to invoke a particular trem. This is conceivably useful. This also suggests that today we're going to be interested in using type systems to prevent situations that aren't inherently stuck states. What is a stuck state other than a state that the type theorist doesn't like?

We're also going to be building our formalism in a slightly different direction than is traditional. Let's make it possible to treat fragments of syntax as values.

1 Programs write programs

Let's start by making the syntax more cumbersome.

$$\begin{array}{rcl} M, N & ::= & x \mid \lambda x{:}A.N \mid N\ N \\ A & ::= & \mathbf{ok} \mid A \to A \\ \Gamma, \Psi & ::= & \cdot \mid \Gamma, x{:}\tau \end{array}$$

Type judgements still work:

$$\begin{gathered}\cdot, x{:}A \to B, y{:}A \vdash x\ y : B \\ \cdot \vdash \lambda x{:}A \to B.\lambda y{:}A.x\ y : (A \to B) \to A \to B\end{gathered}$$

Consider the $x\ y$ in there. You could see those λs as an attempt to extract that term, and make it typeable in an empty enviroment. But we want something slightly different: we want to establish a "higher language level" that talks about open terms.

$$\begin{array}{rcl} M, N & ::= & x \mid \lambda x{:}A.N \mid N\ N \mid \boxed{\mathbf{box}\,(\Psi.N)} \mid \boxed{\mathbf{letbox}\,(N, u.N)} \\ A & ::= & \mathbf{ok} \mid A \to A \mid \boxed{[\Psi]A} \\ \Gamma, \Psi & ::= & \cdot \mid \Gamma, x{:}\tau \\ \Delta & ::= & \boxed{\cdot \mid \Delta, u{::}A[\Psi]} \end{array}$$

Although variables x and metavariables u are distinguished, this language does not distinguish between different levels of *terms*. In my opinion, this is a pain. One consequence of this, incidentally, is that you have to worry about reducing the code that you're manipulating into a normal form. This is like having an overzealous macro system that refuses to handle code until it's ironed out all of the obvious redexes.

But let's check out the type rules:

$$\frac{\Delta; \Gamma, x{:}A \vdash M : B}{\Delta; \Gamma \vdash \lambda x{:}A.M : A \to B}\ \textsc{Lambda} \qquad \frac{\Delta; \Gamma \vdash M : A \to B \qquad \Delta; \Gamma \vdash N : A}{\Delta; \Gamma \vdash M\ N : B}\ \textsc{App}$$

$$\frac{\Gamma(x) = A}{\Delta; \Gamma \vdash x : A}\ \textsc{Var}$$

$$\frac{\Delta; \Psi \vdash M : A}{\Delta; \Gamma \vdash \mathbf{box}\,(\Psi.M) : [\Psi]A}\ \textsc{Box} \qquad \frac{\Delta, u{::}A[\Psi]; \Gamma \vdash N : B \qquad \Delta; \Psi \vdash M : A}{\Delta; \Gamma \vdash \mathbf{letbox}\,(M, u.N) : B}\ \textsc{Letbox}$$

The new $\mathbf{box}\,(\Psi.M)$ form allows us to pick up a N out from under a Γ, making terms with free variables well-typable (provided that the term's free variables and their types appear in its $[\Psi]A$ type). Now we can perform the abstraction we wanted:

$$\cdot \vdash \mathbf{box}\,(\cdot, x{:}A \to B, y{:}A.x\ y) : [\cdot, x{:}A \to B, y{:}A]B$$

We also have an elimination form for $[\Psi]A$ types: $\mathbf{letbox}\,(M, u.N)$, but you'll notice that it doesn't work quite like the function application form. It merely delegates the problem by adding a new name, u, to the new meta-environment, Δ. So now we have these metavariables representing terms floating around, but in order to use those terms, we need to get rid of their free variables. This requires more machinery.

$$\begin{array}{rcl} M, N & ::= & x \mid \lambda x{:}A.N \mid N\ N \mid \mathbf{box}\,(\Psi.N) \mid \mathbf{letbox}\,(N, u.N) \mid \boxed{\mathbf{clo}\,(u, \sigma)} \\ A & ::= & \mathbf{ok} \mid A \to A \mid [\Psi]A \\ \Gamma, \Psi & ::= & \cdot \mid \Gamma, x{:}\tau \\ \Delta & ::= & \cdot \mid \Delta, u{::}A[\Psi] \\ \sigma & ::= & \boxed{\cdot \mid \sigma, M/x} \end{array}$$

Since σ is a substitution, we're probably interested in x and M being type-compatible. Here's the judgement $(\Delta; \Gamma \vdash \sigma : \Psi)$ for substitutions being valid:

$$\frac{\Delta; \Gamma \vdash N_1 : A_1 \ldots}{\Delta; \Gamma \vdash (N_1/x_1 \ldots) : (x_1 : A_1 \ldots)}\ \sigma\textsc{-ok}$$

Substitutions will be useful, because they'll let us finally eliminate those troublesome open terms:

$$\frac{\Delta(u) = u{::}A[\Psi] \qquad \Delta; \Gamma \vdash \sigma : \Psi}{\Delta; \Gamma \vdash \mathbf{clo}\,(u, \sigma) : A}\ \textsc{Metavar}$$

1.1 An example

$$\lambda z{:}[y_1{:}C, y_2{:}D]A.\mathbf{letbox}\,(z, u.\mathbf{box}\,(y_3{:}D.\lambda w{:}C.\mathbf{clo}\,(u, [w/y_1, y_3/y_2]))) \quad : \quad [y_1{:}C, y_2{:}D]A \to [y_3{:}D]C \to A$$

This is a function that takes a term (producing type A) with holes y_1 and y_2, and wraps a lambda around it, producing a term with only a y_3 missing. Additionally, it ensures that all the substitutions are typesafe.

1.2 By the way: logic

Let's cross over the Curry-Howard isomorphism and ask what kind of logic this. (Actually, *Contextual modal type theory* starts with the logic and only eventually gets to the programming interpretation.) Instead of talking about expressions of lambdas and what environments they would be well-typed in, the logic talks about expressions of implications and what possible worlds they would be true in. Terms that are well-typed without reference to worlds are "valid" (i.e., true in all possible worlds).

$$\frac{\Delta; \Psi \vdash M : A}{\Delta; \Gamma \vdash \mathbf{box}\,(\Psi.M) : [\Psi]A}\ \textsc{Box}$$

$$\frac{\Delta; \Psi \vdash A\ \mathbf{true}}{\Delta; \Gamma \vdash [\Psi]A\ \mathbf{valid}}\ \textsc{NecessityIntroduction}$$

A box underneath a set of lambdas translates to the extraction of a truth from underneath a series of implications by tagging it with the environment Ψ necessary to make it true.

However, if you work this out, you might find it an unmotivating approach, since all it amounts to is the expression of modus ponens at many different levels and in many different ways.

One advantage of the logical perspective, however, is that it makes it more clear that there is a missed generalization in all this: we might want to talk about metametavariables or metametametavariables or meta^{n}variables, if our situation is sufficiently complicated. (Such situations come up reasonably frequently on paper, at least.) The paper *Multi-level contextual type theory* covers this.

2 What is it good for?

The main contribution of contextual modal type theory is that it allows for the manipulation of syntax in a way that respects α-equivalence. Specifically, α-equivalent inputs to a program will produce α-equivalent outputs.

There are two main applications of this. If you're using a theorem-proving system, and you find yourself writing many lemmas regarding the names in the source code that you're manipulating, you probably want a better representation of your values, something that provides guarantees of name wellbehavedness for free. This work is the solution to your problem.

But even if you don't need proof that your code works, it would still be nice for your code to work. Therefore, anyone who writes a compiler or optimizer or a macro can benefit from α-equivalence-repsecting representations of code. However, there are some more obstacles up ahead.

3 Source code manipulation that works

Using this system to implement a compiler or a macro system sounds like a great idea at first: preserving α-equivalence sounds a lot like hygiene. In fact, it's even a stronger property, in some sense. Traditional Scheme hygiene prevents macros from interfering with each other, but it still allows them to destructure syntax in a way that ignores binding. That can't happen with this system.

But there's a problem. It only works if the only binding construct in your language is lambda. This mostly defeats the purpose of having a macro system in the first place.

We need a system that allows for user-defined binding constructs. That's where FreshML comes in:

$$
\begin{array}{rcl}
v & ::= & () \mid (v, v) \mid \langle x \rangle\ v \mid x \\
p & ::= & () \mid (x, x) \mid \langle x \rangle\ x \\
e & ::= & v \\
& \mid & \textbf{case}\ v\ (p\ e) \ldots \\
& \mid & \textbf{fresh}\ x\ \textbf{in}\ e \\
& \mid & \textbf{if}\ x = x\ \textbf{then}\ e\ \textbf{else}\ e \\
& \mid & \textbf{let}\ x = e\ \textbf{in}\ e \\
& \mid & f(v \ldots) \\
\tau & ::= & \textbf{atom} \mid \textbf{unit} \mid \tau \times \tau \mid \langle \textbf{atom} \rangle\ \tau \\
\vdots & &
\end{array}
$$

You'll notice that this system is a little awkward. For one thing, it doesn't have higher-order functions. This is an unfortunate sacrifice necessary to support the extra power that it has. You'll also notice that there's a limited vocabulary for binding (just $\langle x \rangle\ v$, which binds x in v). This problem is easy to solve: Cαml (*An overview of Cαml*) can be dropped in to provide a richer language of binding forms. But let's not worry about that for now.

In this system, **fresh** corresponds very roughly to **box**, and **case** is approximately **clo**. But there's something hidden in the semantics.

Most importantly, the **case** construct freshens all the names that become free when it destructures a value. If you have the values $\lambda a_4.a_4$ and $\lambda a_6.a_6$, you'll never be able to tell the difference between them. And you'll notice that the previous system didn't let you destructure syntax at all. Great, huh?

Let's for the moment imagine that we're working in a system which only manipulates ordinary lambda terms, so it can interpret $\langle x \rangle\ v$ as $\lambda x.v$. Let's suppose we want to test whether something is syntactically the identity function or not (note that we're not concerned with complicated functions that are semantically equal to the identity function; we're just looking at syntax):

```
case $f$
  ($\langle x \rangle$ $body$
    case $body$
      ($y$   if $x = y$ then true else false)
      (($a, b$)   false))
```

Note that the **fresh** and **case** constructs now use gensym (or something similar) to create new names. Why do we have to do this icky thing instead of what we had before? Well, for one thing, although FreshML doesn't have this capability on its own, Cαml can have n-ary binding terms (like Scheme's let and lambda), and the manual renaming offered style in the previous section only works if the programmer can write each new name explicitly in code.

Does this make anything go wrong? Yes: we no longer take α-equivalent inputs to α-equivalent outputs. To use an extreme example, the program **fresh** x x is a valid one, but it nondeterministically produces new results from no output.

4 Purity!

The language presented in *Static name control for FreshML* solves this problem:

$$\begin{array}{rcl}
v & ::= & () \mid (v,v) \mid \langle x \rangle\ v \mid x \\
p & ::= & () \mid (x,x) \mid \langle x \rangle\ x \\
e & ::= & v \\
 & \mid & \textbf{case}\ v\ (p\ e)\ldots \\
 & \mid & \textbf{fresh}\ x\ \textbf{in}\ e \\
 & \mid & \textbf{if}\ x = x\ \textbf{then}\ e\ \textbf{else}\ e \\
 & \mid & \textbf{let}\ x = e\ \textbf{where}\ C\ \textbf{in}\ e \\
 & \mid & f(v\ldots) \\
\tau & ::= & \textbf{atom} \mid \textbf{unit} \mid \tau \times \tau \mid \langle \textbf{atom} \rangle\ \tau \\
s & ::= & \mathit{freeatoms}(v) \mid \varnothing \mid s \cup s \mid s \cap s \mid \neg s \\
C & ::= & s = \varnothing \mid s \neq \varnothing \mid C \wedge C \\
\vdots & &
\end{array}$$

To fix this, the semantics will need to be modified such that **fresh** and **case** prohibit the escape of names that they generate. So **fresh** $x\ \lambda x.x$ is okay, but **fresh** $x\ (x, x)$ is bad, because whatever symbol is generated will appear unbound in the value that is returned. This means more to prove to get soundness.

The bad news is that an ordinary type system can no longer prove soundness with this new notion of stuckness, since it has to do with the flow of values.

This is where C comes in. By annotating certain parts of code (including function declarations, which I've left out of this presentation) with extra information, it becomes possible to give the typechecker the hints it needs to show that all the free atoms that will appear in values produced by an expression were produced by "legitimate" means. Joining these things up requires a general-purpose tool, like an SMT solver.

Here's an example of a type rule (now called a "proof rule") from this system:

$$\frac{\Delta, x \vdash \{H \wedge \{x\} \notin \mathit{freeatoms}(\Delta)\}\ \ e\ \ \{P \wedge \{x\} \notin \mathit{freeatoms}(\cdot)\}}{\Delta \vdash \{H\}\ \ \textbf{fresh}\ x\ \textbf{in}\ e\ \ \{P\}}\ \textsc{Fresh}$$

The good news is that it's not necessary to throw the entire program into the SMT solver; just information that links one part of a program to another.

5 Conclusion

The moral of this story isn't a very uplifting one: sometimes the property that you're interested in is non-local, and it takes some heavy type machinery to provide assurances.

6 Bibliography

- *Contextual modal type theory.* Nanevski, Pfenning, Pientka. Introduces a type theory that distinguishes *truth* of propositions in an environment Γ from the *validity* of propositions in an environment Δ. These are connected by the existence of valid propositions asserting that other propositions would be true, given some environment Γ. Crosses the Curry-Howard isomorphism to introduce a language with the corresponding type theory: a language in which terms can be given types contingient upon being placed in a certain environment. Metavariables, distinct from variables, are used to denote these "incomplete" terms.

  Significance: This is an early theoretical treatment of the problem of working with open terms. Although it isn't as pragmatically-oriented as later works, it has the characteristic features of this line of work: a non-extensible homogeneous language in which all terms used as values are kept in normal form.

- *Multi-level contextual type theory.* Boespflug, Pientka. Extends *Contextual modal type theory* by abstracting over the degree of meta-ness of variables. It is possible to have a term with variables representing holes in a variety of different phases. Only a single environment is used, and the environment

is ordered by meta-level. The important invariant is that a variable of level n can only depend on a variable of level m if $n \leq m$.

Significance: I'm not aware of any direct applications of this, but it solves the awkward 2-levelness of the previous work by allowing for meta^n-variables. It's geared towards theorem-proving and proof assistant applications: the authors describe a relatively simple proof assistant task which involves variables from three different levels. Although all three levels conceptually have different meanings, the problem of avoiding free variables and hygienically treating names is the same in any language.

- *Programming with binders and indexed data types.* Cave, Pientka. Builds a dependently-typed language around contextual modal type theory. It uses the trick of indexed types to ensure the termination of typechecking in a dependently-typed language: only a terminating language of terms is allowed to actually appear in types. This system is fairly powerful, and can be viewed as an extension of the Beluga language.

  Significance: This is an example of the convergence of logics-with-support-for-binding and practical dependently-typed languages. The motivation for this is that many dependently-typed languages manipulate an awful lot of source code, and proving lemmas about well-behavedness of names in ASTs is as essential as proving lemmas about the commutivity of operations in arithmetic, and a whole lot harder. Improving the representation for structures with names can help solve that problem.

- *Static name control for FreshML.* Pottier. Describes FreshML, a (first-order) ML-like language which supports ASTs with binding as a core feature, which uses automated renaming to prevent the unhygienic equating of names from different binders. Argues for and adds "purity", a further restriction on the escape of freshened names, which ensures that α-equivalent inputs are taken to α-equivalent outputs. In order to prove this non-escape *statically*, localized SMT-solving, assisted by programmer annotations, must be brought in.

  Significance: This work restores the strong guarantees of contextual type theories in a language oriented towards practical traditional metaprogramming. User-defined binding constructs (described by the type system) are permitted, and terms are manipulated without worrying about normalizing them, or even what the semantics of the object language is. This makes macros that introduce new binding constructs (a very important kind) possible to express.

- *An overview of Cαml.* Pottier. Describes a binding specification language which allows for more complicated binding forms. In FreshML, only a single binder name may be bound by a binding construct, but in Cαml, it is legal to describe binding constructs which harvest names from structures of unknown size (an example of this is the n-arity of `let`), which can even be tree-shaped, and to specify places in a construct in which references are not bound to the names (an example is the right-hand sides of a `let`, but not a `letrec`.)

  Significance: This more powerful system is a drop-in replacement for the one in pure FreshML, and it provides the power to express more interesting and powerful binding forms, such as some members of the `let` family.