A Fix for Dynamic Scope

Ravi Chugh
University of California, San Diego
rchugh@cs.ucsd.edu

Abstract

When the semantics for pure lambda-terms is defined with *dynamic* scope, instead of the more conventional lexical scope, recursive computations can be expressed directly without the need for a `fix` operator, either as a native function or as a lambda-term. In this report, we sketch the design of a type system for a dynamically scoped lambda-calculus based on a simple idea called *open types*. An open function type declares assumptions about the types of free variables in the function body, and these assumptions must be satisfied in every context where the function is used.

We then discuss how the same mechanism can describe well-typed recursive computations in the *lexically* scoped lambda-calculus extended with references. In this setting, recursion can be achieved without a `fix` operator using a well-known encoding called "backpatching." But whereas the typical encoding requires that the reference to a recursive function be initialized to a dummy function of the desired type, open function types allow type checking the backpatching pattern *without* such initialization, an idiom that appears in scripting languages like JavaScript.

Dynamic Scope and Recursion

In a previous era, a language designer would make a concerted choice about the semantics of scope, in particular, whether a free variable should resolve in its enclosing *lexical* environment or the *dynamic* environment in which it is evaluated. Although lexical scope has scored a decisive victory because it enables modular design, we consider a dynamically scoped, pure, lambda-calculus λ_{dyn} not for its merit for program design but for the relative ease with which it allows recursive functions to be defined.

For example, consider the factorial function `fact` in Figure 1, defined with a (non-recursive) `let`-binding instead of `letrec`. Under a lexically scoped semantics, the call to `fact` results in a "variable-not-found" error at run-time because the function body refers to `fact`, which is not defined in the enclosing scope. But with dynamic scope, the occurrence of `fact` in the function body resolves to the function value being defined, thereby enabling recursion without an explicit `fix` operator. We define the syntax and semantics of λ_{dyn} in Figure 2. The rules E-Fun and E-App codify dynamic scope: a "bare" lambda evaluates to itself and the body of a function is evaluated in the environment E in which it is called.

Saving the three characters "`rec`" compared to the definition of `fact` in a lexically scoped language is nothing to write home about. Only slightly more interesting is the definition of mutually recursive functions `tick` and `tock` in Figure 1. Rather than simultaneously defining the functions in a `letrec`, each component is defined separately, and the occurrences of `tick` and `tock` in each function body dynamically resolves to the other function.

More interesting is that a new, quieter version of `tock` can shadow the original, and the subsequent call `tick 2` witnesses the recursion between `tick` and the new version of `tock`. This sort of "late packaging" of mutually recursive functions is impossible with lexical scope; to encode a similar program fragment, two `letrec` definitions are required and the definition of `tick` must be duplicated.

```
let fact n = if n < 1 then 1 else n * fact (n-1) in
fact 5;

let tick n = if n > 0 then "tick " ++ tock n in
let tock n = "tock " ++ tick (n-1) in
tick 2;   // "tick tock tick tock "

let tock n = tick (n-1) in
tick 2;   // "tick tick "
```

Figure 1. Recursion with dynamic scope.

Syntax and Semantics $\boxed{E \vdash e \Downarrow v}$

$$\begin{array}{rcll} e & ::= & c \mid \lambda x.e \mid x \mid e_1\, e_2 & \text{Expressions} \\ & \mid & \texttt{let}\ x = e_1\ \texttt{in}\ e_2 & \\ v & ::= & c \mid \lambda x.e \mid err & \text{Values} \\ E & ::= & E, x \mapsto v \mid - & \text{Evaluation Environments} \end{array}$$

$$\text{[E-Const]}\ \frac{}{E \vdash c \Downarrow c} \qquad \text{[E-Var]}\ \frac{}{E \vdash x \Downarrow E(x)} \qquad \text{[E-Fun]}\ \frac{}{E \vdash \lambda x.e \Downarrow \lambda x.e}$$

$$\text{[E-Delta]}\ \frac{E \vdash e_1 \Downarrow c \quad E \vdash e_2 \Downarrow v_2 \quad \delta(c, v_2) = v}{E \vdash e_1\, e_2 \Downarrow v} \qquad \text{[E-App]}\ \frac{E \vdash e_1 \Downarrow \lambda x.e \quad E \vdash e_2 \Downarrow v_2 \quad E, x \mapsto v_2 \vdash e \Downarrow v}{E \vdash e_1\, e_2 \Downarrow v}$$

$$\text{[E-Let]}\ \frac{E \vdash e_1 \Downarrow v_1 \quad E, x \mapsto v_1 \vdash e_2 \Downarrow v_2}{E \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow v_2}$$

Constants c include primitive values and operators interpreted by a δ-function and assigned types by $ty(c)$. We assume the standard encoding for if-expressions, and we omit rules for evaluating to err. To compare to dynamic scope, the following rules encode lexical scope:

$$\frac{}{E \vdash \lambda x.e \Downarrow (\lambda x.e, E)} \qquad \frac{E \vdash e_1 \Downarrow (\lambda x.e, E') \quad E \vdash e_2 \Downarrow v_2 \quad E', x \mapsto v_2 \vdash e \Downarrow v}{E \vdash e_1\, e_2 \Downarrow v}$$

Figure 2. Syntax and semantics for λ_{dyn}.

Open Types

We propose a type system for λ_{dyn} in Figure 3 designed to prevent the usual set of run-time errors. Because of dynamic scope, no function (even non-recursive ones) can make any unchecked assumptions about the types of free variables it refers to. Accordingly, the central concept in our system is that of an *open* function type, which lists assumptions about the types of all free variables that the function uses.

For example, the following function in our system

$$\lambda x.\, x > y \;::\; (y\!:\!\mathit{Int}) \Rightarrow \mathit{Int} \rightarrow \mathit{Bool}$$

is assigned a type which says that it may be called *only* in environments where y binds an integer, regardless of whether y binds an integer (if it is even defined at all!) when the function is defined. Delaying the checks for all free variables until each calling context seems like sleight of hand; we could instead rewrite every function in λ_{dyn} to abstract over all of its free variables as additional parameters and then use simple function types instead. For example, the previous function would become:

$$\lambda x.\, \lambda y.\, x > y \;::\; \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Bool}$$

But the rewriting approach does not allow us to assign types to recursive functions.

Using open function types, however, we can assign the following types to our examples from before:

$$\begin{array}{rcl} \gamma_1 & \overset{\circ}{=} & \mathit{fact} : (\mathit{fact}\!:\!\mathit{Int} \rightarrow \mathit{Int}) \Rightarrow \mathit{Int} \rightarrow \mathit{Int}, \\ & & \mathit{tick} : (\mathit{tock}\!:\!\mathit{Int} \rightarrow \mathit{Str}) \Rightarrow \mathit{Int} \rightarrow \mathit{Str}, \\ & & \mathit{tock} : (\mathit{tick}\!:\!\mathit{Int} \rightarrow \mathit{Str}) \Rightarrow \mathit{Int} \rightarrow \mathit{Str} \end{array}$$

The T-FUN rule assigns an open function type $(\Gamma) \Rightarrow T_1 \rightarrow T_2$ by type checking the function in the environment Γ; the open type environment γ in which the function is defined is *irrelevant*. In a sense, the T-FUN generalizes the following rule for recursive functions in lexically scoped languages

$$\frac{\Gamma, f\!:\!T_1 \rightarrow T_2 \vdash e :: T_1 \rightarrow T_2}{\Gamma \vdash \mathtt{fix}\,\lambda f.e :: T_1 \rightarrow T_2}$$

to allow *arbitrary* bindings to be assumed during type checking, not just the recursive function being defined. In this way, we can type check mutually recursive functions "one piece at a time."

To type check a function call $f\ e$, the T-APP rule checks that the constraints of all open function types in the environment are mutually satisfied. For this, the Guarantee(γ) operation strips the environment of all assumptions on free variables (recording the most recent binding of a variable if there are multiple), and Rely(γ) collects all of the assumptions. The environment γ is *consistent* if the former is a superset of the latter. For example, the environment γ_1 is consistent because Guarantee(γ_1) = Rely(γ_1) = Γ, where

$$\Gamma = \{\mathit{fact}\!:\!\mathit{Int} \rightarrow \mathit{Int}, \mathit{tick}\!:\!\mathit{Int} \rightarrow \mathit{Str}, \mathit{tock}\!:\!\mathit{Int} \rightarrow \mathit{Str}\}.$$

On the other hand, the following environment is inconsistent:

$$\begin{array}{rcl} \gamma_2 & \overset{\circ}{=} & f : (x\!:\!\mathit{Int}) \Rightarrow \mathit{Int} \rightarrow \mathit{Int}, \\ & & g : (x\!:\!\mathit{Bool}) \Rightarrow \mathit{Bool} \rightarrow \mathit{Bool}, \\ & & x : \mathit{Int} \\ \text{Guarantee}(\gamma_2) & = & \{f\!:\!\mathit{Int} \rightarrow \mathit{Int}, g\!:\!\mathit{Bool} \rightarrow \mathit{Bool}, x\!:\!\mathit{Int}\} \\ \text{Rely}(\gamma_2) & = & \{x\!:\!\mathit{Int}, x\!:\!\mathit{Bool}\} \end{array}$$

For simplicity, the T-APP rule requires that the entire environment is consistent. Instead, to be more permissive, we could check that just the assumptions Γ (and their transitive dependencies) that the function f depends on are satisfied.

We intend that the type system satisfies the following soundness property, but we have not proved it: if $\vdash e :: S$ then $\not\vdash e \Downarrow \mathit{err}$.

Open Types for Backpatching

Although using open function types to describe recursive computations seems interesting, we have already mentioned that dynamic scope is not desirable. But the same idea is also relevant for the lexically scoped lambda-calculus extended with *references* (*i.e.* mutable variables). Figure 4 defines versions of `tick` and `tock` in in this setting, where the mutable variables are initialized to the dummy value `null` and then overwritten with appropriate function definitions. This pattern is quite similar to the "backpatching" encoding of recursion without `fix`,[1] but that encoding requires that the initial values be dummy functions of the appropriate types, because reference types are invariant.

In a system that combines *strong updates*[2] with open function types, however, we can imagine assigning the following types

$$\begin{array}{l} \mathit{tick} : (\mathit{tock}\!:\!\mathit{Ref}\ (\mathit{Int} \rightarrow \mathit{Str})) \Rightarrow \mathit{Ref}\ (\mathit{Int} \rightarrow \mathit{Str}) \\ \mathit{tock} : (\mathit{tick}\!:\!\mathit{Ref}\ (\mathit{Int} \rightarrow \mathit{Str})) \Rightarrow \mathit{Ref}\ (\mathit{Int} \rightarrow \mathit{Str}) \end{array}$$

which allows dummy initializers as long as the assumptions are satisfied by the time the functions are called. We plan to explore this connection with our prior work on static types for JavaScript,[3] since definitions of `tick` and `tock` in JavaScript essentially translate[4] to the code in Figure 4.

Type Checking $\boxed{\gamma \vdash e :: S}$

$$\begin{array}{rcll} T & ::= & B \mid T_1 \rightarrow T_2 & \text{Types} \\ S & ::= & (\Gamma) \Rightarrow T & \text{Open Types} \\ \Gamma & ::= & \Gamma, x\!:\!T \mid - & \text{Type Environments} \\ \gamma & ::= & \gamma, x\!:\!S \mid - & \text{Open Type Environments} \end{array}$$

$$\frac{}{\gamma \vdash c :: ty(c)}\ [\text{T-CONST}] \qquad \frac{}{\gamma \vdash x :: \gamma(x)}\ [\text{T-VAR}]$$

$$\frac{\gamma \vdash e_1 :: S_1 \qquad \gamma, x\!:\!S_1 \vdash e_2 :: S_2}{\gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 :: S_2}\ [\text{T-LET}]$$

$$\frac{\Gamma, x\!:\!T_1 \vdash e :: T_2}{\gamma \vdash \lambda x.e :: (\Gamma) \Rightarrow T_1 \rightarrow T_2}\ [\text{T-FUN}]$$

$$\frac{\gamma(f) = (\Gamma) \Rightarrow T_1 \rightarrow T_2 \qquad \gamma \vdash e :: T_1 \qquad \vdash \gamma}{\gamma \vdash f\ e :: T_2}\ [\text{T-APP}]$$

Environment Consistency $\boxed{\vdash \gamma}$

$$\frac{\text{Guarantee}(\gamma) \supseteq \text{Rely}(\gamma)}{\vdash \gamma}$$

$$\begin{array}{rcl} \text{Guarantee}(-) & = & \emptyset \\ \text{Guarantee}(\gamma, x\!:\!(\Gamma) \Rightarrow T) & = & \text{Guarantee}(\gamma) \uplus \{x\!:\!T\} \\ \text{Rely}(-) & = & \emptyset \\ \text{Rely}(\gamma, x\!:\!(\Gamma) \Rightarrow T) & = & \text{Rely}(\gamma) \cup \Gamma \end{array}$$

$$\Gamma \uplus \{x\!:\!T\} \;=\; \{\, y\!:\!T' \mid y\!:\!T' \in \Gamma \wedge x \neq y \,\} \cup \{x\!:\!T\}$$

Figure 3. Type checking for λ_{dyn}.

```
let (tick, tock) = (ref null, ref null) in
tick := \n. if n > 0 then "tick " ++ !tock n;
tock := \n. "tock " ++ !tick (n-1);
!tick 2; // "tick tock tick tock "
```

Figure 4. Recursion with lexical scope and references.

[1] `www.seas.upenn.edu/~cis500/cis500-f06/lectures/1025.pdf`

[2] Alias Types (Smith *et al.*, ESOP 2000)

[3] Dependent Types for JavaScript (Chugh *et al.*, OOPSLA 2012)

[4] The Essence of JavaScript (Guha *et al.*, ECOOP 2010)