A Type Theory for Linking

(extended abstract)

Michael Arntzenius
advised by Karl Crary

April 24, 2012

1 Audience

This paper is written assuming a working knowledge of the basics of programming language syntax and semantics. Backus-Naur Form (BNF) is used to define grammars and natural-deduction style inference rules are used to define logical judgments. Familiarity with typed λ-calculi, in particular features such as product, universal, and recursive types, is assumed. The Curry-Howard correspondence is employed without remark throughout. Other concepts, such as modal logic, the adjoint calculus, and hereditary substitution, are explained before use. Prior knowledge of these is not necessary.

2 Motivation

Most formal literature on programming languages accounts for three processes: parsing, typechecking and execution. Even the ubiquitous process of compilation generally takes us outside the formal realm; the target language is typically untyped, and arguing that compilation preserves the source language semantics is done informally if at all. Fortunately, there is active research in this area, in the form of verified compilers, typesafe assembly languages, and proof-carrying code. However, there are two processes almost as ubiquitous as compilation that have been mostly ignored by formal efforts: linking and loading.

We define the λ^{lib}-calculus, showing how a typed language with explicit support for linking and loading may be formalized and implemented. Aside from the straightforward goal of filling a gap in our formal understanding of computation, we are particularly motivated by the need of ConcertOS, a project to build a typesafe operating system, for a model of linking and loading. The content of this work is, however, not specific to ConcertOS.

3 Background

3.1 Modal logic and mobility

We initially expected to use some variety of *modal logic* as our typesystem. Modal logics are a family of logics which deal with the concept of *necessity*[1]. "Necessity" is understood in terms of "possible worlds". In general, a proposition may be true in one possible world and false in another.

[1] And its dual, *possibility*, which we do not consider here

"Necessary" propositions are those true in *all* possible worlds. In contrast, "contingent" truths may only be true in "our" world; or at least we only *know* they are true in our world.[2]

Modal logic adds a unary propositional connective expressing that a proposition τ is "necessary", which we write $\Box\tau$. What suggested modal logic to us was that the $\Box$ operator has been shown to represent *mobile* types [2]. The notion of mobile code arises in distributed computation. In general, code in a distributed system may depend on the resources of a particular computational node to run. "Mobile" code is code that can run at any node. Modal logic models this if we let our "possible worlds" be nodes: a necessary proposition is true in every world, just as mobile code can run on any node. $\Box\tau$ is then the type of a piece of mobile code that returns a value of type τ.

Libraries are akin to mobile code: they are stored on-disk in a format portable to any machine with the same operating system and instruction set—at least, as long as it can supply the library's dependencies. This is the sticking point: modal logic provides no good way to represent libraries with dependencies. Explaining this problem requires a digression.

3.2 Dependencies

Suppose that $\Box\tau$ is the type of a library with *no* dependencies that contains code of type τ. What then is the type of a library that *does* have a dependency? First, note that the primary operation on a library with a dependency is linking it against a library fulfilling that dependency.[3] Second, libraries with dependencies are defined in terms of the depended-upon thing, as if it were a free variable. Together these suggest that libraries with dependencies can be viewed as a kind of function, taking their dependency as an argument, and returning their own contents as a result.

Consider then a library l that depends on a τ_1 and produces a τ_2. If libraries are functions of their dependencies, perhaps we can give l the type $\Box\tau_1 \to \Box\tau_2$? Unfortunately this does not suffice, because it is not *mobile*. $\Box\tau_1 \to \Box\tau_2$ is an ordinary function type, and its values are not necessarily of a form that can be written to disk and shipped between machines the way a library (even a library with as-yet unfilled dependencies) can.

Can we fix this by adding a surrounding $\Box$, giving $l : \Box(\Box\tau_1 \to \Box\tau_2)$? Unfortunately, this still does not suffice. While this type is mobile, it still uses an ordinary function as the mechanism for linking against a dependency, which is undesirable for two reasons. First, it means that linking, which involves calling the function contained in l, can have arbitrary side-effects (eg. nontermination or malicious behavior), which is clearly undesirable.

Second, it makes *partial linking*, where we link a library against just one of its several dependencies, impractical. Consider that curried functions must receive their arguments *in order*; given $f : \tau_1 \to \tau_2 \to \tau_3$ and $y : \tau_2$, I cannot apply f to y without also having some $x : \tau_1$. I can get around this by creating a wrapper function:

$$\lambda x : \tau_1.\ f\ x\ y$$

However, this merely delays the actual call to f until the first argument x is received.[4] If f is the underlying function of a library with two dependencies, this means we cannot do the actual work of linking the library against its second dependency until we have its first. Real linkers do not suffer from this problem.

[2] This is an extremely rough characterization, and ignores many important varieties of modal logic.

[3] There is also the issue of mutually dependent libraries. For simplicity, and because we believe that it is in practice quite feasible to avoid such circular dependencies, we ignore this problem entirely.

[4] Uncurried functions $f : \tau_1 \times \tau_2 \to \tau_3$ exhibit a stronger version of this problem: they cannot be called until *all* arguments are received.

3.3 The adjoint calculus

To deal with dependencies, we turn to Benton and Wadler's *adjoint calculus* [1]. Adjoint calculi are a family of λ-calculi that can be used to "encode" both modal logic and other logics such as intuitionistic linear logic. Here we concern ourselves only with its relation to modal logic.

The adjoint calculus "splits" modal logic into two layers, an "upper" and a "lower". The upper layer corresponds to necessary or mobile things—for us, libraries. The lower layer corresponds to ordinary, contingent things—for us, an ordinary programming language to which we wish to add explicit support for libraries. Thus we split our types into upper-layer library interfaces I and lower-layer types τ; and our terms into libraries L and ordinary terms e. In Figure 1 we give the syntax and relevant rules of our version of the adjoint calculus, which we proceed to develop into the λ^{lib}-calculus.

Syntax:

$$\begin{array}{rrcl@{\qquad}rrcl} \text{library contexts} & \Delta & ::= & \cdot \mid \Delta, u : I & \text{term contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : \tau \\ \text{interfaces} & I & ::= & \lceil \tau \rceil \mid \ldots & \text{types} & \tau & ::= & \lfloor I \rfloor \mid \ldots \\ \text{libraries} & L & ::= & u \mid \mathsf{code}\, e \mid \ldots & \text{terms} & e & ::= & x \mid \mathsf{lib}\, L \mid \mathsf{use}\, L \\ & & & & & & \mid & \mathsf{load}\, u = e_1\, \mathsf{in}\, e_2 \mid \ldots \end{array}$$

Judgments: $\Delta \vdash L : I$ and $\Delta; \Gamma \vdash e : \tau$

Rules:

$$\frac{\Delta; \cdot \vdash e : \tau}{\Delta \vdash \mathsf{code}\, e : \lceil \tau \rceil}\ \lceil \mathrm{I} \rceil \qquad \frac{\Delta \vdash L : \lceil \tau \rceil}{\Delta; \Gamma \vdash \mathsf{use}\, L : \tau}\ \lceil \mathrm{E} \rceil$$

$$\frac{\Delta \vdash L : I}{\Delta; \Gamma \vdash \mathsf{lib}\, L : \lfloor I \rfloor}\ \lfloor \mathrm{I} \rfloor \qquad \frac{\Delta; \Gamma \vdash e_1 : \lfloor I \rfloor \quad \Delta, u : I; \Gamma \vdash e_2 : \tau}{\Delta; \Gamma \vdash \mathsf{load}\, u = e_1\, \mathsf{in}\, e_2 : \tau}\ \lfloor \mathrm{E} \rfloor$$

Figure 1: Syntax and typing rules of an adjoint calculus for libraries

In place of the singular modal operator $\Box$, we have two operators. The first, $\lceil \tau \rceil \in I$, projects from the term to the library layer; it is the interface of a library containing code of type τ. The second, $\lfloor I \rfloor$, projects from the library to the term layer; it is the type of a run-time reference to a library with interface I. Thus the modal $\Box\tau$ becomes $\lfloor \lceil \tau \rceil \rfloor$: the type of a run-time reference to a library containing code of type τ.

As we have two layers, we also have two contexts: Δ for library variables and Γ for term variables. Terms may depend on libraries, so the typing judgment $\Delta; \Gamma \vdash e : \tau$ for terms involves both contexts. However, the typing judgment $\Delta \vdash L : I$ for libraries lacks a term context, preventing libraries from depending on arbitrary run-time values. Thus to embed a library in a program via the lib operator, we must throw away our Γ context, as in the $\lfloor \mathrm{I} \rfloor$ rule. Similarly, to embed code into a library, it must typecheck with an empty term context, as in the $\lceil \mathrm{I} \rceil$ rule. The $\lceil \mathrm{E} \rceil$ rule is straightforward: if we have a library $L : \lceil \tau \rceil$ containing code of type τ, we can use it within a program to retrieve the embedded τ. $\lfloor \mathrm{E} \rfloor$ is only slightly more subtle: given $e_1 : \lfloor I \rfloor$, a reference to a library with interface I, we can bind a library variable $u : I$ to the library it refers to to.

3.3.1 Extending the library layer

The only library-layer operator *needed* to encode modal $\Box$ is $\lceil \tau \rceil$. But having separated the two layers, we are free to add other connectives to I. It is this additional expressiveness that suits the

adjoint calculus to our purposes: library-layer connectives let us express the the "super-structure" of a library, beyond merely the type of the code it contains. In particular, we can express dependencies as *library-layer* functions, distinct from ordinary term-layer functions:

$$I ::= ... \mid I \to I$$
$$L ::= ... \mid \lambda u : I.\, L \mid L\ L$$

This cleanly separates the library-level computation of linking a library against its dependencies from ordinary run-time function application. We can thus avoid arbitrary link-time side-effects simply by not introducing any side-effectful operations to the library layer. Partial linking, as we shall see, is more complicated, but still doable.

In general, by adding library layer connectives, we express that these correspond to the structure of the library *itself*, not the code it contains. For a concrete example of this distinction, let us first add library products:

$$I ::= ... \mid I \times I$$
$$L ::= ... \mid \langle L, L \rangle \mid \pi_i\, L$$

Now, consider the following C-language header files:

File: `A.h`	*File:* `B.h`
`int x;` `int y;`	`struct { int x; int y; } p;`

A library implementing the interface expressed by `A.h` will contain two labels, each of type `int`. A library implementing `B.h` will contain one label of type `struct {int x; int y;}` (in other words, a pair of `int`s). The interface of `A.h` is thus best represented by $\lceil \mathsf{int} \rceil \times \lceil \mathsf{int} \rceil$ (involving a library-layer product), while `B.h`'s interface is $\lceil \mathsf{int} \times \mathsf{int} \rceil$ (involving a term-layer product).

3.3.2 Polymorphism

Before presenting the full λ^{lib}-calculus, one complication remains. The adjoint calculus we have presented so far lacks parametric polymorphism, the ability to universally quantify over types. Adding support for this is straightforward syntactically, but requires adding a new context Ψ for type variables, which raises the question of how this context behaves in our two-layer system. Is Ψ discarded like Γ when we move into the library layer, or preserved like Δ? For the sake of expressivity, we'd like Ψ to be preserved, otherwise it becomes impossible to implement a function with a type such as $\forall \alpha.\, \lfloor \lceil \alpha \rceil \rfloor \to \alpha$. Since types τ are conceptually lower-layer things, however, it's not immediately obvious that this is safe. Luckily for us, it turns out that it is.[5]

4 The λ^{lib}-calculus

The λ^{lib} calculus is a polymorphic adjoint calculus with functions and products at the library layer, and functions, universals, and recursive types at the term layer. The library-layer connectives we have discussed already; the term-layer type operators were chosen to make the term layer Turing-complete with a minimum of bother, the better to model real programming languages and so serve as a proof of concept. The syntax and typing rules are given in Figure 2.

[5] Explanation and proof to be given in full thesis.

Syntax:

$$
\begin{array}{rcl}
I & ::= & \lceil \tau \rceil \mid I \to I \mid I \times I \\
\tau & ::= & \lfloor I \rfloor \mid \tau \to \tau \mid \alpha \mid \forall \alpha.\, \tau \mid \mu \alpha.\, \tau \\
L & ::= & u \mid \lambda u : I.\, L \mid L\ L \mid \langle L, L \rangle \mid \pi_i\, L \mid \mathsf{code}\, e \\
e & ::= & x \mid \lambda x : \tau.\, e \mid e\ e \mid \Lambda \alpha.\, e \mid e[\tau] \mid \mathsf{roll}_{\mu \alpha.\, \tau}\, e \mid \mathsf{unroll}\, e \\
 & \mid & \mathsf{lib}\, L \mid \mathsf{use}\, L \mid \mathsf{load}\ u = e\ \mathsf{in}\ e
\end{array}
$$

Judgments:

$$
\begin{array}{ll}
\Psi \vdash I\ \mathsf{iface} & \Psi; \Delta \vdash L : I \\
\Psi \vdash \tau\ \mathsf{type} & \Psi; \Delta; \Gamma \vdash e : \tau \qquad e\ \mathsf{valish}
\end{array}
$$

Rules:

$$
\frac{}{\Psi, \alpha, \Psi' \vdash \alpha\ \mathsf{type}} \qquad \frac{\Psi \vdash I\ \mathsf{iface}}{\Psi \vdash \lfloor I \rfloor\ \mathsf{type}} \qquad \frac{\Psi \vdash \tau_1\ \mathsf{type} \quad \Psi \vdash \tau_2\ \mathsf{type}}{\Psi \vdash \tau_1 \to \tau_2\ \mathsf{type}}
$$

$$
\frac{\Psi, \alpha \vdash \tau\ \mathsf{type}}{\Psi \vdash \forall \alpha.\, \tau\ \mathsf{type}} \qquad \frac{\Psi, \alpha \vdash \tau\ \mathsf{type}}{\Psi \vdash \mu \alpha.\, \tau\ \mathsf{type}}
$$

$$
\frac{\Psi \vdash \tau\ \mathsf{type}}{\Psi \vdash \lceil \tau \rceil\ \mathsf{iface}} \qquad \frac{\Psi \vdash I_1\ \mathsf{iface} \quad \Psi \vdash I_2\ \mathsf{iface}}{\Psi \vdash I_1 \to I_2\ \mathsf{iface}} \qquad \frac{\Psi \vdash I_1\ \mathsf{iface} \quad \Psi \vdash I_2\ \mathsf{iface}}{\Psi \vdash I_1 \times I_2\ \mathsf{iface}}
$$

$$
\frac{}{\Psi; \Delta, u : I, \Delta' \vdash u : I}
$$

$$
\frac{\Psi; \Delta, u : I_1 \vdash L : I_2}{\Psi; \Delta \vdash \lambda u : I_1.\, L : I_1 \to I_2} \qquad \frac{\Psi; \Delta \vdash L_1 : I_1 \to I_2 \quad \Psi; \Delta \vdash L_2 : I_1}{\Psi; \Delta \vdash L_1\ L_2 : I_2}
$$

$$
\frac{\Psi; \Delta \vdash L_1 : I_1 \quad \Psi; \Delta \vdash L_2 : I_2}{\Psi; \Delta \vdash \langle L_1, L_2 \rangle : I_1 \times I_2} \qquad \frac{\Psi; \Delta \vdash L : I_1 \times I_2}{\Psi; \Delta \vdash \pi_i\, L : I_i}
$$

$$
\frac{e\ \mathsf{valish} \quad \Psi; \Delta; \cdot \vdash e : \tau}{\Psi; \Delta \vdash \mathsf{code}\, e : \lceil \tau \rceil} \qquad \frac{\Psi; \Delta \vdash L : \lceil \tau \rceil}{\Psi; \Delta; \Gamma \vdash \mathsf{use}\, L : \tau}
$$

$$
\frac{}{\Psi; \Delta; \Gamma, x : \tau, \Gamma' \vdash x : \tau}
$$

$$
\frac{\Psi; \Delta; \Gamma, x : \tau_1 \vdash e : \tau_2}{\Psi; \Delta; \Gamma \vdash \lambda x : \tau_1.\, e : \tau_1 \to \tau_2} \qquad \frac{\Psi; \Delta; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Psi; \Delta; \Gamma \vdash e_2 : \tau_1}{\Psi; \Delta; \Gamma \vdash e_1\ e_2 : \tau_2}
$$

$$
\frac{\Psi, \alpha; \Delta; \Gamma \vdash e : \tau}{\Psi; \Delta; \Gamma \vdash \Lambda \alpha.\, e : \forall \alpha.\, \tau} \qquad \frac{\Psi \vdash \tau'\ \mathsf{type} \quad \Psi; \Delta; \Gamma \vdash e : \forall \alpha.\, \tau}{\Psi; \Delta; \Gamma \vdash e[\tau'] : [\tau'/\alpha]\, \tau}
$$

$$
\frac{\Psi; \Delta; \Gamma \vdash e : [\mu \alpha.\, \tau/\alpha]\, \tau}{\Psi; \Delta; \Gamma \vdash \mathsf{roll}_{\mu \alpha.\, \tau}\, e : \mu \alpha.\, \tau} \qquad \frac{\Psi; \Delta; \Gamma \vdash e : \mu \alpha.\, \tau}{\Psi; \Delta; \Gamma \vdash \mathsf{unroll}\, e : [\mu \alpha.\, \tau/\alpha]\, \tau}
$$

$$
\frac{\Psi; \Delta \vdash L : I}{\Psi; \Delta; \Gamma \vdash \mathsf{lib}\, L : \lfloor I \rfloor} \qquad \frac{\Psi; \Delta \vdash L : \lceil \tau \rceil}{\Psi; \Delta; \Gamma \vdash \mathsf{use}\, L : \tau} \qquad \frac{\Psi; \Delta; \Gamma \vdash e_1 : \lfloor I \rfloor \quad \Psi; \Delta, u : I; \Gamma \vdash e_2 : \tau}{\Psi; \Delta; \Gamma \vdash \mathsf{load}\ u = e_1\ \mathsf{in}\ e_2 : \tau}
$$

$$
\frac{}{\lambda x : \tau.\, e\ \mathsf{valish}} \qquad \frac{}{\Lambda \alpha.\, e\ \mathsf{valish}} \qquad \frac{e\ \mathsf{valish}}{\mathsf{roll}_{\mu \alpha.\, \tau}\, e\ \mathsf{valish}} \qquad \frac{}{\mathsf{lib}\, L\ \mathsf{valish}}
$$

Figure 2: Syntax and typing rules of the λ^{lib}-calculus

4.1 Suspensions versus values

Careful inspection of Figure 2 reveals the unusual judgment e valish. This is meant to convey that e is close enough to being a value as makes no difference. The troubling rule here (the reason we don't just call it e value) is the axiom $\mathsf{lib}\, L$ valish, which puts no constraints on the form of L. We'll see why this is acceptable later; for now, e valish can be read as if it said "e is a value".

e valish notably makes an appearance as a premise of the $\lceil \mathrm{I} \rceil$ rule:

$$\frac{e\ \mathsf{valish} \quad \Psi; \Delta; \cdot \vdash e : \tau}{\Psi; \Delta \vdash \mathsf{code}\, e : \lceil \tau \rceil}$$

In contrast with the adjoint calculus presented so far, the λ^{lib}-calculus requires that in the library $\mathsf{code}\, e$, the enclosed term e be a value. Without this constraint, we are forced into an ugly choice: either evaluating a library term L may involve evaluating embedded terms e, bringing back the danger of unrestricted side-effects during linking; or, we must understand $\mathsf{code}\, e$ as a *suspension* that delays evaluating e until it is extracted via use. The former is clearly unacceptable. The latter is a principled solution, but fails to capture the semantics of real-world libraries. We therefore add the e valish restriction, on the grounds that it better models real-world semantics at a neglible cost to expressivity.

4.2 Partial linking and hereditary substitution

As yet we have not presented dynamic semantics for the λ^{lib}-calculus. The primary issue we face in constructing such a semantics is accounting for partial linking: the ability to link a library with multiple dependencies against any one of them independently. Since we are representing libraries as functions of their dependencies, this reduces to the problem of *partial evaluation*: reducing a function applied to some known and some unknown arguments to a function of only the unknown arguments.

Partial evaluation is usually considered in the context of compiler optimization, but unfortunately we do not have the luxury of treating it as a mere optimization. Luckily, there is a standard technique for achieving this, known as *hereditary substitution*. Hereditary substitution can intuitively be thought of as keeping all terms "as normalized as possible given their free variables" at all times. As such, all the actual work of reduction in a system of hereditary substitution occurs during substitution—when we eliminate free variables.

Hereditary substitution requires separating the terms being dealt with (in our case, libraries) into "canonical" and "atomic" forms. We give the splitting of libraries L into canonical M and atomic R in Figure 3, and the corresponding changes required in other parts of the λ^{lib}-calculus syntax. Canonical forms consist of introduction forms ($\lambda u : I.\ M$, $\langle M, M \rangle$, $\mathsf{code}\, e$) and embedded atomic forms ($\mathsf{at}\ R$). Atomic forms consist of a series of elimination forms ($R\ M$, $\pi_i\, M$) applied to a "head" variable (u). Directly replacing this head variable with a canonical term, as in naïve substitution, would be syntactically invalid. Thus, only irreducible uses of elimination forms are expressible. Hereditary substitution is the algorithm by which we replace variables without violating this syntactic constraint, and its rules are also given in Figure 3.

The translation from the original λ^{lib}-calculus into the form needed for hereditary substitution can be accomplished without much difficulty[6], and it is this translation that justifies the laxness of the judgment e valish. Post-translation, $\mathsf{lib}\, L$ becomes $\mathsf{lib}\, M$, and thanks to the syntactic restrictions of hereditary substitution, M is guaranteed not to require further reduction (modulo containing some free variables). In light of this we rename the judgment to e value.

[6] Translation to be given in full thesis.

Modified syntax:

$$\begin{array}{rcl} M & ::= & \mathsf{at}\, R \mid \lambda u\!:\!I.\, M \mid \langle M, M\rangle \mid \mathsf{code}\, e \\ R & ::= & u \mid R\, M \mid \pi_i\, R \\ e & ::= & \ldots \mid \mathsf{lib}\, M \mid \mathsf{use}\, R \end{array}$$

Modified typing judgments:

$\Psi; \Delta \vdash L : I$ splits into $\Psi; \Delta \vdash M : I$ and $\Psi; \Delta \vdash R : I$

e valish is replaced by e value

Modified typing rules:

$$\frac{\Psi; \Delta \vdash M : I}{\Psi; \Delta; \Gamma \vdash \mathsf{lib}\, M : \lfloor I \rfloor} \qquad \frac{\Psi; \Delta \vdash R : \lceil \tau \rceil}{\Psi; \Delta; \Gamma \vdash \mathsf{use}\, R : \tau}$$

$$\frac{\Psi; \Delta \vdash R : I}{\Psi; \Delta \vdash \mathsf{at}\, R : I} \qquad \frac{}{\Psi; \Delta, u : I, \Delta' \vdash u : I} \qquad \frac{e\ \mathsf{value} \quad \Psi; \Delta; \cdot \vdash e : \tau}{\Psi; \Delta \vdash \mathsf{code}\, e : \lceil \tau \rceil}$$

$$\frac{\Psi; \Delta, u : I_1 \vdash M : I_2}{\Psi; \Delta \vdash \lambda u\!:\!I_1.\, M : I_1 \to I_2} \qquad \frac{\Psi; \Delta \vdash R : I_1 \to I_2 \quad \Psi; \Delta \vdash M : I_1}{\Psi; \Delta \vdash R\, M : I_2}$$

$$\frac{\Psi; \Delta \vdash M_1 : I_1 \quad \Psi; \Delta \vdash M_2 : I_2}{\Psi; \Delta \vdash \langle M_1, M_2\rangle : I_1 \times I_2} \qquad \frac{\Psi; \Delta \vdash R : I_1 \times I_2}{\Psi; \Delta \vdash \pi_i\, R : I_i}$$

$$\frac{}{\lambda x\!:\!\tau.\, e\ \mathsf{value}} \qquad \frac{}{\Lambda \alpha.\, e\ \mathsf{value}} \qquad \frac{}{\mathsf{roll}_{\mu\alpha.\,\tau}\ \mathsf{value}} \qquad \frac{}{\mathsf{lib}\, M\ \mathsf{value}}$$

Substitution judgments: $[M/u]\, M \to M$, $[M/u]\, R \to M$, $[M/u]\, e \to e$

Substitution rules:

$$\frac{[M/u]\, R \to N}{[M/u]\, \mathsf{at}\, R \to N} \qquad \frac{[M/u]\, e \to e'}{[M/u]\, \mathsf{code}\, e \to \mathsf{code}\, e'}$$

$$\frac{[M/u]\, N \to N' \quad (u \neq v)}{[M/u]\, \lambda v\!:\!I.\, N \to \lambda v\!:\!I.\, N'} \qquad \frac{[M/u]\, M_1 \to M_1' \quad [M/u]\, M_1 \to M_2'}{[M/u]\, \langle M_1, M_2\rangle \to \langle M_1', M_2'\rangle}$$

$$\frac{}{[M/u]\, u \to M} \qquad \frac{(u \neq v)}{[M/u]\, v \to \mathsf{at}\, v}$$

$$\frac{[M/u]\, R \to \mathsf{at}\, R' \quad [M/u]\, N \to N'}{[M/u]\, R\, N \to \mathsf{at}\, R'\, N'} \qquad \frac{[M/u]\, R \to \mathsf{at}\, R'}{[M/u]\, \pi_i\, R \to \pi_i\, R'}$$

$$\frac{[M/u]\, R \to \lambda v\!:\!I.\, O \quad [M/u]\, N \to N' \quad [N/v]\, O \to O'}{[M/u]\, R\, N \to O'} \qquad \frac{[M/u]\, R \to \langle M_1, M_2\rangle}{[M/u]\, \pi_i\, R \to M_i}$$

$$\frac{[M/u]\, N \to N'}{[M/u]\, \mathsf{lib}\, N \to \mathsf{lib}\, N'} \qquad \frac{[M/u]\, R \to \mathsf{at}\, R'}{[M/u]\, \mathsf{use}\, R \to \mathsf{use}\, R'} \qquad \frac{[M/u]\, R \to \mathsf{code}\, e}{[M/u]\, \mathsf{use}\, R \to e}$$

(All other $[M/u]\, e \to e$ rules just distribute the substitution across subexpressions.)

Figure 3: Modified λ^{lib}-calculus using hereditary substition

4.3 Dynamic semantics

Having defined hereditary substitution we can proceed to define a full dynamic semantics for the modified λ^{lib}-calculus, given in Figure 4.

Judgments: $e \mapsto e$

Rules:

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2} \qquad \frac{e_1\ \mathsf{value} \quad e_2 \mapsto e_2'}{e_1\ e_2 \mapsto e_1\ e_2'} \qquad \frac{}{(\lambda x : \tau.\ e_1)\ e_2 \mapsto [e_2/x]\, e_1}$$

$$\frac{e_1 \mapsto e_1'}{\langle e_1, e_2 \rangle \mapsto \langle e_1', e_2 \rangle} \qquad \frac{e_1\ \mathsf{value} \quad e_2 \mapsto e_2'}{\langle e_1, e_2 \rangle \mapsto \langle e_1, e_2' \rangle} \qquad \frac{e \mapsto e'}{\pi_i\ e \mapsto \pi_i\ e'} \qquad \frac{e_1\ \mathsf{value} \quad e_2\ \mathsf{value}}{\pi_i\ \langle e_1, e_2 \rangle \mapsto e_i}$$

$$\frac{e \mapsto e'}{e\,[\tau] \mapsto e'\,[\tau]} \qquad \frac{}{\Lambda\alpha.\ e\,[\tau] \mapsto [\tau/\alpha]\, e}$$

$$\frac{e \mapsto e'}{\mathsf{roll}_{\mu\alpha.\,\tau}\ e \mapsto \mathsf{roll}_{\mu\alpha.\,\tau}\ e'} \qquad \frac{e \mapsto e'}{\mathsf{unroll}\ e \mapsto \mathsf{unroll}\ e'} \qquad \frac{e\ \mathsf{value}}{\mathsf{unroll}\ (\mathsf{roll}_{\mu\alpha.\,\tau}\ e) \mapsto e}$$

$$\frac{e_1 \mapsto e_1'}{\mathsf{load}\ u = e_1\ \mathsf{in}\ e_2 \mapsto \mathsf{load}\ u = e_1'\ \mathsf{in}\ e_2} \qquad \frac{[M/u]\, e_2 \to e_2'}{\mathsf{load}\ u = \mathsf{lib}\ M\ \mathsf{in}\ e_2 \mapsto e_2'}$$

Figure 4: Dynamic semantics of λ^{lib}-calculus with hereditary substitution

Proofs of progress and preservation will be given in full thesis.

5 CAM$^{\mathsf{lib}}$

The "Categorical Abstract Machine", or CAM, is a simple abstract machine used as a compilation target for λ-calculi and other functional languages. We extend a simplified version of the CAM with instructions that implement our hereditary-substitution library operations. Efficient implementation requires careful use of explicit substitutions to avoid deep-copying. Details will be in full thesis.

6 Implementation

We have implemented, in Standard ML, a typechecker and compiler for the λ^{lib}-calculus, translating through the λ^{lib}-calculus with hereditary substitution, targetting the CAM$^{\mathsf{lib}}$. We have also implemented a bytecode interpreter for the CAM$^{\mathsf{lib}}$ (written in C). Link to source code and examples will be included in full thesis. Current work may be found at `https://github.com/rntz/ttol`.

References

[1] N. Benton and P. Wadler. Linear logic, monads, and the lambda calculus. In *11th Annual IEEE Symposium on Logic in Computer Science*, 1996.

[2] Tom Murphy, VII. *Modal Types for Mobile Code.* PhD thesis, Carnegie Mellon, January 2008. Available as technical report CMU-CS-08-126.