A Type Theory for Linking

(rough draft)

Michael Arntzenius
advised by Karl Crary

May 17, 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 [4]. 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, impossible. 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 achieve 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, $\lceil\tau\rceil \in I$, injects terms into the library layer; it is the interface of a library containing code of type τ. The second, $\lfloor I \rfloor$, injects libraries into 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 code into a library via the code operator (the $\lceil\mathrm{I}\rceil$ rule), it must typecheck with an empty term context; and when embedding a library in a program via the lib operator (the $\lfloor\mathrm{I}\rfloor$ rule), we throw away our Γ context. 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 $e_1 : \lfloor I \rfloor$, which evaluates to 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 type operator *needed* to encode modal $\Box$ is $\lceil\tau\rceil$. But having separated the two layers, we are free to add other operators to I. It is this expressiveness that suits the adjoint calculus to our purposes: library-layer operators let us express the the "super-structure" of a library, beyond merely the type of the code it contains, in a way that modal logic cannot. 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 (*link-time*) computation of linking a library against its dependencies from ordinary (*run-time*) function application. We can thus avoid arbitrary link-time side-effects by simply not introducing any side-effectful operations to the library layer. Partial linking, as we shall see, is more complicated, but still feasible.

In general, by adding library layer operators, 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 polymorphism introduces a new context Ψ for type variables (see Figure 2), which raises the question of how this context should behave in our two-layer system: Is Ψ discarded like Γ when we move into the library layer, or preserved like Δ? That is, is the typing judgment for libraries of the form $\Psi; \Delta \vdash L : I$ or does it remain $\Delta \vdash L : I$?

For the sake of expressivity, we'd like to choose the former and preserve Ψ in the library layer, 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 it turns out that it is.

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 operators we have

$$\begin{array}{rrcl}
\text{type contexts} & \Phi & ::= & \cdot \mid \Phi, \alpha \\
\text{types} & \tau & ::= & ... \mid \alpha \\
\text{terms} & e & ::= & ... \mid \Lambda\alpha.\, e \mid e[\tau]
\end{array}$$

Figure 2: Adding polymorphism to the adjoint calculus

discussed already; the term operators were chosen to make the term layer Turing-complete with a minimum of bother, the better to model a real programming language and so serve as a proof of concept. The syntax and typing rules are given in Figure 3.

4.1 Suspensions versus values

Careful inspection of Figure 3 reveals the use of e value as a premise of the $\lceil \mathrm{I} \rceil$ rule:

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

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 to make 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 take $\mathsf{code}\, e$ to be 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 value restriction, on the grounds that it better models real-world semantics at a neglible cost to expressivity.

4.2 Dynamic semantics, partial linking, and hereditary substitution

As yet we have not presented dynamic semantics for the λ^{lib}-calculus. The primary difficulty 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 separates terms into "canonical" and "atomic" forms. In our case, we separate libraries L into canonical M and atomic R. We present the resulting $\lambda^{\mathsf{lib}}_{\mathsf{hs}}$-calculus. Canonical libraries M are either introduction forms ($\lambda u : I.\, M$, $\langle M, M \rangle$, $\mathsf{code}\, e$) or embedded atomic forms ($\mathsf{at}\ R$). Atomic forms R 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.

Syntax:

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

Judgments:

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

Rules:

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{}{u\;\mathsf{value}} \qquad \frac{}{\lambda u\!:\!I.\,L\;\mathsf{value}} \qquad \frac{L_1\;\mathsf{value} \quad L_2\;\mathsf{value}}{\langle L_1, L_2\rangle\;\mathsf{value}} \qquad \frac{e\;\mathsf{value}}{\mathsf{code}\;e\;\mathsf{value}}$$

Figure 3: Syntax and typing rules of the λ^{lib}-calculus

Hereditary substitution is the algorithm by which we replace variables without violating this syntactic constraint, and its rules are given along with the dynamic semantics for the $\lambda^{\mathsf{lib}}_{\mathsf{hs}}$-calculus in Figure 5. Since the function of hereditary substitution is to keep libraries in canonical form, there is no need for a separate "evaluation" judgment in the dynamic semantics for libraries. Instead of typical progress and preservation lemmas, therefore, we have substitution lemmas (at least for the library layer):

Lemma 1 (Substitution of Types)**.**

1. *(a) If* $\Psi \vdash \tau$ *type and* $\Psi, \alpha, \Psi' \vdash \tau'$ *type then* $\Psi, \Psi' \vdash [\tau/\alpha]\,\tau'$ *type.*
   (b) If $\Psi \vdash \tau$ *type and* $\Psi, \alpha, \Psi' \vdash I$ *iface then* $\Psi, |psi' \vdash [\tau/\alpha]\,I$ *iface.*
2. *(a) If* $\Psi \vdash \tau$ *type and* $\Psi, \alpha, \Psi'; \Delta \vdash M : I$ *then* $\Psi, \Psi'; \Delta \vdash [\tau/\alpha]\,M : [\tau/\alpha]\,I$*.*
   (b) If $\Psi \vdash \tau$ *type and* $\Psi, \alpha, \Psi'; \Delta \vdash R : I$ *then* $\Psi, \Psi'; \Delta \vdash [\tau/\alpha]\,R : [\tau/\alpha]\,I$*.*
   (c) If $\Psi \vdash \tau$ *type and* $\Psi, \alpha, \Psi'; \Delta; \Gamma \vdash e : \tau'$ *then* $\Psi, \Psi'; \Delta; \Gamma \vdash [\tau/\alpha]\,e : [\tau/\alpha]\,\tau'$*.*

Lemma 2 (Substitution of Terms)**.**
If $\Psi; \Delta; \Gamma \vdash e : \tau$ *and* $\Psi; \Delta; \Gamma, x{:}\tau, \Gamma' \vdash e' : \tau'$ *then* $\Psi; \Delta; \Gamma, \Gamma' \vdash [e/x]\,e' : \tau'$*.*

Theorem 1 (Progress)**.** *If* $\cdot; \cdot; \cdot \vdash e : \tau$ *then either* e *value or* $e \mapsto e'$*.*

Theorem 2 (Preservation)**.** *If* $\cdot; \cdot; \cdot \vdash e : \tau$ *and* $e \mapsto e'$ *then* $\cdot; \cdot; \cdot \vdash e' : \tau$*.*

Theorem 3 (Substitution of libraries)**.** *If* $\Psi; \Delta \vdash M : I$*, then:*

1. *If* $\Psi; \Delta, u{:}I, \Delta'; \Gamma \vdash e : \tau$*, then* $[M/u]\,e \to e'$ *and* $\Psi; \Delta, \Delta'; \Gamma \vdash e' : \tau$*. Moreover, if* e *value then* e' *value.*
2. *If* $\Psi; \Delta, u{:}I, \Delta' \vdash N : I'$ *then* $[M/u]\,N \to N'$ *and* $\Psi; \Delta, \Delta' \vdash N' : I'$*.*
3. *If* $\Psi; \Delta, u{:}I, \Delta' \vdash R : I$ *then* $[M/u]\,R \to N$ *and* $\Psi; \Delta, \Delta' \vdash N : I'$ *and either* $N = \mathsf{at}\,R'$ *for some* R' *or* $I' \sqsubseteq I$ *(where* $\sqsubseteq$ *is structural subsumption).*

Proofs will be available online at `https://github.com/rntz/ttol`.

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

The "Categorical Abstract Machine", or CAM, is a simple abstract machine used as a compilation target for λ-calculi and other functional languages [2]. 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.

5.1 Instruction set

We begin with a simplified presentation of (a variant of) the CAM due to Xavier Leroy [3]. Our initial CAM is an abstract machine whose state is a three-tuple $\langle c, e, s\rangle$ of an instruction sequence c, an environment e, and a stack s, defined by:

$$\begin{array}{rcll} \text{code} & c & ::= & i;\, c \mid \mathsf{Return} \\ \text{environment} & e & ::= & v.\, e \mid \cdot \\ \text{stack} & s & ::= & v.\, s \mid \langle c, e\rangle.\, s \mid \cdot \\ \text{instructions} & i & ::= & \mathsf{Access}(n) \mid \mathsf{Closure}(c) \mid \mathsf{Apply} \\ \text{values} & v & ::= & [c, e] \\ \text{natural numbers} & n & & \end{array}$$

Syntax:

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

Judgments:

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

L value disappears.

Rules:

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

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

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

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

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

Translation from λ^{lib}: We define an operation $\ulcorner\_\urcorner$ that takes libraries L to canonical libraries M. We use library substitution, defined in Figure 5.

$$\begin{aligned} \ulcorner u \urcorner &= \mathsf{at}\, u \\ \ulcorner \lambda u\!:\!I.\, L \urcorner &= \lambda u\!:\!I.\, \ulcorner L \urcorner \\ \ulcorner \langle L_1, L_2\rangle \urcorner &= \langle \ulcorner L_1 \urcorner, \ulcorner L_2 \urcorner \rangle \\ \ulcorner L_1\ L_2 \urcorner &= \begin{cases} M' & \text{if } \ulcorner L_1 \urcorner = \lambda u\!:\!I.\, M \text{ and } [\ulcorner L_2 \urcorner / u]\, M \to M' \\ \mathsf{at}\, (R\, \ulcorner L_2 \urcorner) & \text{if } \ulcorner L_1 \urcorner = \mathsf{at}\, R \end{cases} \\ \ulcorner \pi_i\, L \urcorner &= \begin{cases} M_i & \text{if } \ulcorner L \urcorner = \langle M_1, M_2 \rangle \\ \mathsf{at}\, (\pi_i\, R) & \text{if } \ulcorner L \urcorner = \mathsf{at}\, R \end{cases} \end{aligned}$$

Figure 4: Syntax & typing rules of $\lambda^{\mathsf{lib}}_{\mathsf{hs}}$ (where they differ from λ^{lib}) and translation from λ^{lib}

Judgments:

Library-layer: $[M/u]\,M \to M,\;\; [M/u]\,R \to M,\;\; [M/u]\,e \to e$

Term-layer: $e \mapsto e$

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.)

Evaluation 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 5: Dynamic semantics of $\lambda_{\mathsf{hs}}^{\mathsf{lib}}$

The transitions of our simplified CAM are as follows:

State before			**State after**		
Code	*Env*	*Stack*	*Code*	*Env*	*Stack*
Access(n); c	$v_0. \ldots v_n. e$	s	c	$v_0. \ldots v_n. e$	$v_n. s$
Closure(c'); c	e	s	c	e	$[c', e]. s$
Apply; c	e	$v. [c', e']. s$	c'	$v. e'$	$\langle c, e\rangle. s$
Return	e	$v. \langle c', e'\rangle. s$	c'	e'	$v. s$

Informally, the environment e is a list of values, used to store local variables. The stack s stores temporary result values v and return frames $\langle c, e\rangle$. Code c is some sequence of instructions terminated by a Return, which (naturally) returns the value on the top of the stack to the return frame below it. Our only values are closures $[c, e]$ of code and a captured environment. The Access(n) instruction accesses the n^{th} value in the environment (ie. some local variable); Closure(c) creates a closure of c with the current environment; and Apply applies a function.

A simple translation $\mathcal{T}(\_)$ from an untyped lambda calculus using DeBruijn-indexed variables into CAM code c may be defined as follows:

$$\begin{array}{rcl} e & ::= & n \mid \lambda.\, e \mid e\; e \\ \\ \mathcal{T}(e) & = & \mathcal{C}(e,\, \mathsf{Return}) \\ \mathcal{C}(n,\, c) & = & \mathsf{Access}(n);\; c \\ \mathcal{C}(\lambda.\, e,\, c) & = & \mathsf{Closure}(\mathcal{T}(e));\; c \\ \mathcal{C}(e_1\; e_2,\, c) & = & \mathcal{C}(e_1,\, \mathcal{C}(e_2,\, \mathsf{Apply};\; c)) \end{array}$$

Intuitively speaking, all that is necessary to extend this CAM to handle the $\lambda^{\mathsf{lib}}_{\mathsf{hs}}$-calculus is to extend values v with libraries M, extend the state with a "library environment" σ representing bound library variables (in the same sense as e represents bound term variables), and add instructions to handle our lib, load, and use constructs. We do this in Figure 6.

However, to implement hereditary substitution efficiently we need a variant of *explicit substitution*, as will be explained in 5.2. From this concern comes much added complexity, so the significance of the definitions in Figure 6 may not be immediately apparent.

5.2 Explicit substitutions and copying concerns

The CAM$^{\text{lib}}$, somewhat unusually for an abstract machine, manipulates entities at runtime that are essentially abstract syntax trees—namely, libraries. This seems unavoidable, as no efficient compilation strategy is known for hereditary substitution, unlike ordinary substitution (for which the CAM's environment-and-closure approach implements one such strategy). This should not be too surprising: the operational semantics for our library layer are designed to provide the same functionality as an ordinary linker and loader. In real systems, the information needed to load a library or link it against something else is not compiled into executable instructions residing within the library itself, but is given as metadata to linkers and loaders which manipulate it. On our view, then, linkers and loaders are nothing more than *interpreters* for the instructions contained in libraries' metadata—which is exactly what we have produced in the CAM$^{\text{lib}}$.

Having resigned ourselves to performing capture-avoiding substitution at run-time, then, we are faced with the problem of how to minimize the expense of this operation. The usual solution is to represent variables by their DeBruijn indices: a variable is simply a natural number indicating how many enclosing binders to "skip over" before we find the binder for that variable. For example, $\lambda x.\, \lambda y.\, x\; y$ becomes $\lambda.\, \lambda.\, 1\; 0$ in DeBruijn notation. When we substitute under a binder, we must *lift* (increase by 1) free variables inside the term being substituted, so that they still refer to the correct binder. Assuming an operation $\uparrow e$ which performs this *lift*, the equation for substitution under a binder then becomes:

$$[e'/n]\, \lambda.\, e = \lambda.\, [\uparrow e'/n+1]\, e$$

The problem is that the operation $\uparrow e$ requires traversing the entirety of e to find and increment free variables. This is expensive both in time and, less obviously, in *space*: unless we are careful, the entire term e will be copied once for each binder we substitute under. This is especially worrying when the terms we are substituting are in fact *libraries*: libraries are potentially very large objects. Moreover, traversing and copying them is not only expensive in itself, but unnecessary duplication of the code contained in libraries can unnecessarily enlarge programs linked against them, resulting in reduced spatial locality and worse cache performance.

Luckily there is a well-known solution to this problem, namely *explicit substitutions*. Much like our use of hereditary substitution as a method of partial evaluation, explicit substitution is not

Syntax:

$$
\begin{array}{rrcl}
\text{natural numbers} & n & & \\
\text{code} & c & ::= & i;\, c \mid \mathsf{Return} \\
\text{term environment} & e & ::= & v.\, e \mid \cdot \\
\text{library substitution} & \sigma & ::= & \mathcal{M}.\, \sigma \mid \uparrow^{n} \\
\text{stack} & s & ::= & v.\, s \mid \langle c, e, \sigma\rangle .\, s \mid \cdot \\
\text{instructions} & i & ::= & \mathsf{Access}(n) \mid \mathsf{Closure}(c) \mid \mathsf{Apply} \\
 & & \mid & \mathsf{Func}(n,\, c) \mid \mathsf{Lib}(\mathcal{M}) \mid \mathsf{Use}(\mathcal{R}) \mid \mathsf{Load} \\
\text{values} & v & ::= & [c, e, \sigma] \mid \mathsf{lib}\, \mathcal{M} \\
\text{canonical libraries} & \mathcal{M} & ::= & \mathcal{M}\uparrow^{n} \mid \mathsf{at}\, \mathcal{R} \mid \lambda .\, \mathcal{M} \mid \langle M, M\rangle \mid \mathsf{code}\, (c) \mid \mathsf{code}\, (\mathsf{lib}\, \mathcal{M}) \\
\text{atomic libraries} & \mathcal{R} & ::= & \mathcal{R}\uparrow^{n} \mid 0 \mid \mathcal{R}\, \mathcal{M} \mid \pi_i\, \mathcal{R}
\end{array}
$$

Operations:

$$
\begin{array}{rcll}
\mathsf{code}^{-1}\, (\mathcal{M}) & \to & v & \text{code extraction} \\
\mathsf{code}^{-1}\, (\mathsf{code}\, (c)) & = & [c, \cdot, \uparrow^{0}] & \\
\mathsf{code}^{-1}\, (\mathsf{code}\, (\mathsf{lib}\, \mathcal{M})) & = & \mathsf{lib}\, \mathcal{M} & \\
 & & & \\
\sigma \circ \sigma & \to & \sigma & \text{substitution composition} \\
\uparrow^{n} \circ \uparrow^{m} & = & \uparrow^{(n+m)} & \\
\uparrow^{0} \circ \sigma & = & \sigma & \\
\uparrow^{n+1} \circ (\mathcal{M}.\, \sigma) & = & \uparrow^{n} \circ \sigma & \\
(\mathcal{M}.\, \sigma) \circ \sigma' & = & [\sigma']\, \mathcal{M}.\, (\sigma \circ \sigma') &
\end{array}
$$

See Figure 7 for the definitions of substitution on libraries $[\sigma]\, \mathcal{M} \to \mathcal{M}$, atoms $[\sigma]\, \mathcal{R} \to \mathcal{M}$, and code $[\sigma]\, c \to c$.

Transitions:

State before			**State after**		
Code	*Env/Subst*	*Stack*	*Code*	*Env/Subst*	*Stack*
$\mathsf{Access}(n);\, c$	$\langle v_0. \ldots v_n.\, e, \sigma\rangle$	s	c	$\langle v_0. \ldots v_n.\, e, \sigma\rangle$	$v_n.\, s$
$\mathsf{Closure}(c');\, c$	$\langle e, \sigma\rangle$	s	c	$\langle e, \sigma\rangle$	$[c', e, \sigma].\, s$
$\mathsf{Apply};\, c$	$\langle e, \sigma\rangle$	$v.\, [c', e', \sigma'].\, s$	c'	$\langle v.\, e', \sigma'\rangle$	$\langle c, e, \sigma\rangle\, .\, s$
Return	$\langle e, \sigma\rangle$	$v.\, \langle c', e', \sigma'\rangle\, .\, s$	c'	$\langle e', \sigma'\rangle$	$v.\, s$
$\mathsf{Func}(n,\, c');\, c$	$\langle e, \sigma\rangle$	s	c	$\langle e, \sigma\rangle$	$[c', \cdot, \uparrow^{0}].\, s$
$\mathsf{Lib}(\mathcal{M});\, c$	$\langle e, \sigma\rangle$	s	c	$\langle e, \sigma\rangle$	$\mathsf{lib}\, ([\sigma]\, \mathcal{M}).\, s$
$\mathsf{Use}(\mathcal{R});\, c$	$\langle e, \sigma\rangle$	s	c	$\langle e, \sigma\rangle$	$\mathsf{code}^{-1}\, ([\sigma]\, \mathcal{R}).\, s$
$\mathsf{Load};\, c$	$\langle e, \sigma\rangle$	$\mathsf{lib}\, \mathcal{M}.\, s$	c	$\langle e, \mathcal{M}.\, \sigma'\rangle$	s
				where $\sigma' = \mathsf{at}\, 0.\, (\sigma \circ \uparrow^{1})$	

Figure 6: The CAM$^{\text{lib}}$

Substitution on canonical libraries, $[\sigma]\,\mathcal{M} \to \mathcal{M}$:

$$\begin{aligned}
[{\uparrow}^0]\,\mathcal{M} &= \mathcal{M}\\
[\sigma]\,\mathcal{M}{\uparrow}^n &= [{\uparrow}^n \circ \sigma]\,\mathcal{M}\\
[{\uparrow}^n]\,\mathcal{M} &= \mathcal{M}{\uparrow}^n\\
[\sigma]\,\langle \mathcal{M}_1, \mathcal{M}_2\rangle &= \langle [\sigma]\,\mathcal{M}_1, [\sigma]\,\mathcal{M}_2\rangle\\
[\sigma]\,\lambda.\,\mathcal{M} &= \lambda.\,[\mathsf{at}\,0.\,(\sigma \circ {\uparrow}^1)]\,\mathcal{M}\\
[\sigma]\,\mathsf{code}\,(c) &= \mathsf{code}\,([\sigma]\,c)\\
[\sigma]\,\mathsf{code}\,(\mathsf{lib}\,\mathcal{M}) &= \mathsf{code}\,([\sigma]\,\mathcal{M})\\
[\sigma]\,\mathsf{at}\,\mathcal{R} &= [\sigma]\,\mathcal{R}
\end{aligned}$$

Substitution on atomic libraries, $[\sigma]\,\mathcal{R} \to \mathcal{M}$:

$$\begin{aligned}
[{\uparrow}^0]\,\mathcal{R} &= \mathcal{R}\\
[\sigma]\,\mathcal{R}{\uparrow}^n &= [{\uparrow}^n \circ \sigma]\,\mathcal{R}\\
[{\uparrow}^n]\,\mathcal{R} &= \mathcal{R}{\uparrow}^n\\
[\mathcal{M}.\,\sigma]\,0 &= \mathcal{M}\\
[\sigma]\,(\mathcal{R}\;\mathcal{M}) &= \begin{cases} \mathsf{at}\,((\mathcal{R}' \Uparrow n)\;([\sigma]\,\mathcal{M})) & \text{if } [\sigma]\,\mathcal{R} \approx (\mathsf{at}\,\mathcal{R}'){\uparrow}^n\\ [(([\sigma]\,\mathcal{M}).\,{\uparrow}^n]\,\mathcal{M}' & \text{if } [\sigma]\,\mathcal{R} \approx (\lambda.\,\mathcal{M}'){\uparrow}^n \end{cases}\\
[\sigma]\,(\pi_i\,\mathcal{R}) &= \begin{cases} (\mathsf{at}\,(\pi_i\,\mathcal{R}')) \Uparrow n & \text{if } [\sigma]\,\mathcal{R} \approx (\mathsf{at}\,\mathcal{R}'){\uparrow}^n\\ \mathcal{M}_i \Uparrow n & \text{if } [\sigma]\,\mathcal{R} \approx \langle \mathcal{M}_1, \mathcal{M}_2\rangle{\uparrow}^n \end{cases}
\end{aligned}$$

Where $\mathcal{M}_1 \approx \mathcal{M}_2$ iff either $\mathcal{M}_1 = \mathcal{M}_2$ or $\mathcal{M}_1{\uparrow}^0 = \mathcal{M}_2$, and defining $\mathcal{M} \Uparrow n$ and $\mathcal{R} \Uparrow n$ as follows:

$$\begin{aligned}
\mathcal{M} \Uparrow 0 &= \mathcal{M} & \qquad \mathcal{R} \Uparrow 0 &= \mathcal{R}\\
(\mathcal{M}{\uparrow}^n) \Uparrow m &= \mathcal{M}{\uparrow}^{n+m} & \qquad (\mathcal{R}{\uparrow}^n) \Uparrow m &= \mathcal{R}{\uparrow}^{n+m}\\
\mathcal{M} \Uparrow n &= \mathcal{M}{\uparrow}^n & \qquad \mathcal{R} \Uparrow n &= \mathcal{R}{\uparrow}^n
\end{aligned}$$

Substitution on instructions sequences, $[\sigma]\,c \to c$:

$$\begin{aligned}
[{\uparrow}^0]\,c &= c\\
[\sigma]\,\mathsf{Return} &= \mathsf{Return}\\
[\sigma]\,\mathsf{Load};\,c &= \mathsf{Load};\;[\mathsf{at}\,0.\,(\sigma \circ {\uparrow}^1)]\,c\\
[\sigma]\,\mathsf{Lib}(\mathcal{M});\,c &= \mathsf{Lib}([\sigma]\,\mathcal{M});\;[\sigma]\,c\\
[\sigma]\,\mathsf{Use}(\mathcal{R});\,c &= \begin{cases} \mathsf{Func}(n,\,c');\;[\sigma]\,c & \text{if } [\sigma]\,\mathcal{R} \approx (\mathsf{code}\,(c')){\uparrow}^n\\ \mathsf{Lib}(\mathcal{M} \Uparrow n);\;[\sigma]\,c & \text{if } [\sigma]\,\mathcal{R} \approx (\mathsf{code}\,(\mathsf{lib}\,\mathcal{M})){\uparrow}^n\\ \mathsf{Use}(\mathcal{R}' \Uparrow n);\;[\sigma]\,c & \text{if } [\sigma]\,\mathcal{R} \approx (\mathsf{at}\,\mathcal{R}){\uparrow}^n \end{cases}\\
[\sigma]\,\mathsf{Closure}(c');\,c &= \mathsf{Closure}([\sigma]\,c');\;[\sigma]\,c\\
[\sigma]\,\mathsf{Func}(n,\,c');\,c &= \begin{cases} \mathsf{Func}(n,\,c');\;[\sigma]\,c & \text{if } {\uparrow}^n \circ \sigma = {\uparrow}^n\\ \mathsf{Func}(0,\,[{\uparrow}^n \circ \sigma]\,c');\;[\sigma]\,c & \text{otherwise} \end{cases}\\
[\sigma]\,i;\,c &= i;\;[\sigma]\,c
\end{aligned}$$

Figure 7: Explicit hereditary substitution in the CAM$^{\text{lib}}$

usually thought of as a method for avoiding unnecessary copying. Rather, explicit substitutions express the properties of substitution, usually thought of as a transformation, in an algebraic manner, by adding substitutions as first-class objects to the syntax of a language. For example, a simple lambda calculus with explicit substitution could look something like the following:

$$\begin{array}{rcl} e & ::= & n \mid \lambda.\, e \mid e\ e \mid e[\sigma] \\ \sigma & ::= & e.\, \sigma \mid \uparrow^n \end{array}$$

$$\begin{array}{rcl} n[\uparrow^m] & = & n + m \\ (e_1\ e_2)[\sigma] & = & e_1[\sigma]\ e_2[\sigma] \\ (\lambda.\, e)[\sigma] & = & \lambda.\, e[0.\, (\sigma \circ \uparrow^1)] \end{array} \qquad \begin{array}{rcl} \uparrow^n \circ \uparrow^m & = & \uparrow^{n+m} \\ \uparrow^0 \circ \sigma & = & \sigma \\ \uparrow^{n+1} \circ (e.\, \sigma) & = & \uparrow^n \circ \sigma \\ (e.\, \sigma) \circ \sigma' & = & e[\sigma'].\, (\sigma \circ \sigma') \end{array}$$

These definitions seem abstruse, but produce useful algebraic properties. For example, it is easily shown that $e[\sigma][\sigma'] = e[\sigma \circ \sigma']$, and that $\uparrow^0$ forms a left- and right-identity of the composition operator $\circ$.

The usefulness of explicit substitutions for avoiding copying lies in the fact that they delay work. For example, in the λ-calculus given above, the rule for substituting under a binder is:

$$(\lambda.\, e)[\sigma] = \lambda.\, e[0.\, (\sigma \circ \uparrow^1)]$$

The whole work of performing this translation consists in evaluating the composition $\sigma \circ \uparrow^1$, which in essence shifts every term in the substitution σ up by 1. This performs the "lift" which we needed when using DeBruijn notation. However, the only thing that needs to be traversed to evaluate $\sigma \circ \uparrow^1$ is the substitution σ; we do not need to traverse the terms in it, because to lift a term e by one we can simply return the term $e[\uparrow^1]$.

There are many different formulations of explicit substitution, some of which are even lazier than the one presented here (for example, one can let composition of substitutions $\sigma \circ \sigma$ be itself a form of substitution, rather than a function on substitutions, as given above). However, we cannot simply choose our favorite flavor for use in the CAM$^{\text{lib}}$. For the function of explicit substitutions is to *delay* work, whereas the purpose of hereditary substitution—the whole reason we need to be doing substitutions at runtime—is to be able to partially evaluate, to *not* delay work that can already be done.

However, even in hereditary substitution there is some work it is safe to delay. In particular, the operation of "lifting" an expression when substituting under a binder cannot cause a reduction to occur in the expression being lifted, so it is safe to allow *explicit lifts*. This we do, adding the forms $\mathcal{M}\uparrow^n$ and $\mathcal{R}\uparrow^n$ to the syntax of canonical and atomic libraries in the CAM$^{\text{lib}}$. The definitions in Figures 6 and 7 are what result after working through the requirements of hereditary substitution in our new setting of explicit lifts.

6 Implementation

We have implemented, in Standard ML, a typechecker and compiler for the λ^{lib}-calculus, translating through the $\lambda^{\mathsf{lib}}_{\mathsf{hs}}$-calculus, targetting the CAM$^{\text{lib}}$. We have also implemented, in C, a bytecode interpreter for the CAM$^{\text{lib}}$. The code 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] Cousineau G., Curien P.-L., and Mauny M. The categorical abstract machine. *Science of Computer Programming*, 8:173–202, 1987.

[3] Xavier Leroy. From Krivine's machine to the Caml implementations. Invited talk given at the KAZAM workshop, 2005. URL `http://gallium.inria.fr/~xleroy/talks/zam-kazam05.pdf`.

[4] Tom Murphy, VII. *Modal Types for Mobile Code*. PhD thesis, Carnegie Mellon, January 2008. URL `http://tom7.org/papers/`. Available as technical report CMU-CS-08-126.