Type-safe Distributed Programming with ML5 [⋆]

Tom Murphy VII, Karl Crary, and Robert Harper

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
`tom7,crary,rwh@cs.cmu.edu`

Abstract We present ML5, a high level programming language for spatially distributed computing. The language, a variant of ML, allows an entire distributed application to be developed and reasoned about as a unified program. The language supports transparent mobility of any kind of code or data, but its type system, based on modal logic, statically excludes programs that use mobile resources unsafely. The ML5 compiler produces code for all of the hosts that may be involved in the computation. These hosts may be heterogeneous, with different resources and even different architectures. Currently, our compiler and runtime are specialized to the particular case of web programming: a distributed computation with two sites, the web browser and the web server.

1 Introduction

ML5 is a high-level programming language for distributed computing. The language is designed particularly for those programs that are spatially distributed; where parts of the program must run in physically or logically distinct places. Typically such programs must be distributed because of local resources (such as databases or consoles for interacting with a user) that can only be accessed at those places. ML5's type system permits the programmer to describe the available local resources, and then excludes all programs that use them unsafely.

Distributed applications are often developed by writing a set of programs, one for each host, that communicate via a protocol on network sockets. In contrast, ML5 allows an entire distributed application to be developed as a unified program. This has several benefits: The application may share rich, higher-order data structures, including abstract types, between different hosts. It can even maintain references to arbitrary remote resources, as long as those resources not *used* remotely. More importantly, the program can be reasoned about as a single semantic entity. Reasoning about the behavior of a set of programs communicating via a network can be awkward, particularly when the programs are written in different languages. ML5's dynamic semantics is a straightforward extension of ML's. The compiler can type check the code to statically verify that certain kinds of runtime failure are impossible. ML5's type system is based on modal logic,

[⋆] The ConCert Project is supported by the National Science Foundation under grant ITR/SY+SI 0121633: "Language Technology for Trustless Software Dissemination."

a kind of logic that permits simultaneous reasoning from multiple perspectives. This logical basis means that the features for distributed computing integrate naturally into ML's type system, retaining (for example) type inference.

From the source program the compiler produces code for all of the hosts that may be involved in the computation. These hosts may be heterogeneous, with different sets of available resources and even different architectures.

ML5 is the subject of Murphy's Ph.D. thesis and is still in development—the language does not incorporate some desirable features such as a module system or high-level database integration. Some planned features (such as exceptions and mutual exclusion) are not yet implemented in the compiler. The compiler performs only trivial optimizations, yielding generated code that is somewhat slow. However, the implementation works well enough to run useful demo applications. Our current prototype is specialized to the particular case of web programming: a heterogeneous distributed computation with exactly two sites, the web browser and web server.

We will begin with a brief review of the modal logic IS5 and the particular formulation we use for ML5 (Section 2). We then present ML5's core features using web programming as a source of examples (Section 3). The remainder of the paper discusses the interesting facets of how ML5 is implemented: our marshaling strategy based on type representations and the complications of typed closure conversion in a modal setting (Section 4.1), and the particulars of producing distributed applications for web browsers (Section 4.2). We conclude with a discussion of related, ongoing, and future work (Section 6).

2 Modal Logic IS5

IS5 is a modal logic with the ability to reason about truth from multiple simultaneous perspectives, which are called "possible worlds." These possible worlds arise from contingent assumptions that differ from world to world. In our application of modal logic to distributed computing, the logical worlds correspond to the hosts involved in a computation, and the contingent assumptions to the local resources particular to these hosts.

Various related logics are distinguished by the way in which the possible worlds can access one another; IS5 is a simple degenerate case where every pair of worlds can access one another. Of the several ways to define a modal logic, we find an explicit worlds formulation [16] to be most suitable for our type system [11]. This formulation uses a judgment $\Gamma \vdash A @ \mathrm{w}$ which states that under the assumptions in Γ, the proposition A is true at the world w. Γ holds assumptions of the form $B @ \mathrm{w}'$ (positing B is true at w'), and ω (assuming the existence of a world ω). World expressions w include only these bound world variables ω and world constants, written $\mathbf{w}$. We can only use an assumption $A @ \mathrm{w}$ to conclude that fact at the same world. The standard connectives from intuitionistic logic are expressed by attaching "$@ \mathrm{w}$" everywhere; for instance, implication is

$$\frac{\Gamma, A \,@\, \mathrm{w} \vdash B \,@\, \mathrm{w}}{\Gamma \vdash A \supset B \,@\, \mathrm{w}} \supset\text{-I} \qquad \frac{\Gamma \vdash A \supset B \,@\, \mathrm{w} \quad \Gamma \vdash A \,@\, \mathrm{w}}{\Gamma \vdash B \,@\, \mathrm{w}} \supset\text{-E}$$

Modal logic often focuses on two connectives, $\Box A$ and $\Diamond B$ ("A is true in all worlds;" "B is true in some world"). We find that an explicit worlds formulation gives us the ability to define $\Box$ and $\Diamond$ in terms of finer connectives. The most important of these is the `at` modality (following Jia [6]), which internalizes the @ judgment into a proposition. It is defined by

$$\frac{\Gamma \vdash A \,@\, \mathrm{w}'}{\Gamma \vdash A \,\mathtt{at}\, \mathrm{w} \,@\, \mathrm{w}'} \text{at-I} \qquad \frac{\Gamma \vdash A \,\mathtt{at}\, \mathrm{w}' \,@\, \mathrm{w} \qquad \Gamma, A \,@\, \mathrm{w}' \vdash C \,@\, \mathrm{w}}{\Gamma \vdash C \,@\, \mathrm{w}} \text{at-E}$$

A modal logic where propositions can mention worlds is known as a "hybrid logic" [4]; contrary to its name we find the connective to be central to our logic. For instance, $\Box A$ is definable as $\forall \omega . A \,\mathtt{at}\, \omega$. Typed closure conversion (Section 4.1) makes heavy use of the `at` modality. In contrast, we do not use the $\Box$ or $\Diamond$ connectives in any of our examples.

The final feature of our logic that distinguishes it from other formulations of S5 is our perspective-shifting rule `get`. This rule allows for reasoning at a world w_1 to be nested within reasoning at a world w_2.

$$\frac{\Gamma \vdash A \,@\, \mathrm{w}' \quad A\ \mathsf{mobile}}{\Gamma \vdash A \,@\, \mathrm{w}} \text{get}$$

This rule would be nonsense for arbitrary A: all worlds would then conclude exactly the same facts. The judgment mobile that restricts this rule to certain propositions is better explained in terms of the values of the programming language that characterize those propositions, so we leave that for Section 3. As examples, $A \,\mathtt{at}\, \mathrm{w}$ is mobile for any A, and $A \supset B$ is never mobile.

The `get` rule exists for the benefit of the dynamic semantics. It allows us to isolate all of the communication between hosts into this one rule, ensuring that the other rules avoid any "action at a distance" [11]. For example, without `get`, the `at`-E rule would have to allow the proof of $A \,\mathtt{at}\, w'$ to come from an arbitrary third world. With our decomposition, if we want to use a proof from another world, we explicitly move it first.

Our decompositions preserve the meaning of the logic while allowing for a more natural programming language and implementation. In the next section we describe this programming language.

3 ML5

ML5's syntax and semantics are based on core Standard ML [7]. The largest difference is that ML5's typing judgment is stratified by world, like the truth judgment of IS5. Here, a world is a place in which a computation might run. We type check an expression M using the judgment $\Gamma \vdash M : A @ \mathrm{w}$, which means that under variable bindings Γ, the expression M has type A in the world w. It is best to think of the judgment $M : A @ \mathrm{w}$ as meaning that M is *for* w, rather than *at* w. Although M can only be evaluated at w, at runtime it may be moved around between worlds and placed in data structures at other worlds. World expressions can be either variables ω or constants **w**. Every program must begin execution somewhere; in ML5 this world is the constant **home**. The entire program is therefore also typechecked starting at **home**.

An ML5 program begins by describing what it knows about the universe. This includes the declaration of world constants and the local resources available to them. Here is a working example:[1]

```
extern javascript world home
extern val alert : string -> unit @ home
do alert [Hello, home!]
```

This first line is unnecessary (because the constant `home` is already provided by the compiler) but serves to show how worlds can be declared. The *world-kind* `javascript` dictates that this world (which will be the web browser) runs JavaScript [3] code; this is used only by the backend when generating code for the hosts involved in a program. We can support other worldkinds by implementing a code generator and runtime for them; currently, we support `javascript` and `bytecode` (Section 4.2). The declaration `extern val` asserts the existence of a local resource, in this case, a function called `alert`. The compiled program will expect to find a symbol called `alert` at the world `home`, and a program variable `alert` is bound in the code that follows. It is also possible to declare external abstract types, and global resources (Section 3.1). The `do` declaration evaluates an expression for effect, in this case calling the `alert` function on the supplied string constant. This application type checks because all of these declarations are checked at the world `home`; if `alert` were declared to be at a different world `server` we would not be able to call it without first traveling to the server.

To write distributed programs we also need dynamic tokens with which to refer to worlds. A token for the world w has type w `addr`, and can be thought of as the address of that world. A world can compute its own address with the `localhost()` expression, whose typing rule appears in Figure 1. Typically, a program also expects to know the addresses of other worlds when it begins and imports them with `extern val`. We use an address by traveling to the world it indicates, using the `get` construct. For example, here is a program that involves two worlds, the web browser and server:

```
extern bytecode world server
extern val server : server addr @ home
extern val version : unit -> string @ server
extern val alert : string -> unit @ home
do alert (from server
          get version ())
```

[1] Our examples omit the required syntax for wrapping declarations as compilation units, since the implementation currently only supports a single compilation unit.

$$\frac{}{\Gamma \vdash \mathtt{localhost}() : \mathrm{w}\,\mathtt{addr}\,@\mathrm{w}} \qquad \frac{A\ \mathsf{mobile} \quad \Gamma \vdash M : \mathrm{w}'\,\mathtt{addr}\,@\mathrm{w} \quad \Gamma \vdash N : A@\mathrm{w}'}{\Gamma \vdash \mathtt{from}\ M\ \mathtt{get}\ N : A@\mathrm{w}}$$

$$\frac{\Gamma, \omega' \vdash v : A@\omega'}{\Gamma \vdash \mathtt{sham}\ \omega'.v : \text{☘}_{\omega'}A@\mathrm{w}} \qquad \frac{A\ \mathsf{mobile} \quad \Gamma \vdash M : A@\mathrm{w}}{\Gamma \vdash \mathtt{put}\ u = M \overset{\mathrm{w}}{\rightsquigarrow} u{\sim}A} \qquad \frac{\Gamma \vdash M : \text{☘}_{\omega}A@\mathrm{w}}{\Gamma \vdash \mathtt{valid}\ u = M \overset{\mathrm{w}}{\rightsquigarrow} u{\sim}\omega.A}$$

$$\frac{}{\Gamma, x{:}A, \Gamma'@\mathrm{w} \vdash x : A@\mathrm{w}} \qquad \frac{}{\Gamma, u{\sim}\omega.A, \Gamma' \vdash u : [^{\mathrm{w}}/_{\omega}]A@\mathrm{w}} \qquad \frac{\Gamma \vdash v : A@\mathrm{w}}{\Gamma \vdash \mathtt{hold}\ v : A\ \mathsf{at}\ \mathrm{w}@\mathrm{w}'}$$

$$\frac{\Gamma \vdash d \overset{\mathrm{w}}{\rightsquigarrow} \Gamma' \quad \Gamma, \Gamma' \vdash M : C@\mathrm{w}}{\Gamma \vdash \mathtt{let}\ d\ \mathtt{in}\ M : C@\mathrm{w}} \qquad \frac{\Gamma \vdash M : A\ \mathsf{at}\ \mathrm{w}'@\mathrm{w}}{\Gamma \vdash \mathtt{drop}\ x = M \overset{\mathrm{w}}{\rightsquigarrow} x{:}A@\mathrm{w}'}$$

Figure 1. Some rules from the ML5 internal language, which have been simplified for presentation purposes. The judgment $\Gamma \vdash d \overset{\mathrm{w}}{\rightsquigarrow} \Gamma'$ states that the declaration d, checked at w, produces new hypotheses Γ'.

$$\frac{}{\text{☘}A\ \mathsf{mobile}} \qquad \frac{}{A\ \mathsf{at}\ \mathrm{w}\ \mathsf{mobile}} \qquad \frac{b \in \{\mathtt{string}, \mathtt{int}, \ldots\}}{b\ \mathsf{mobile}} \qquad \frac{\begin{matrix}\alpha\ \mathsf{mobile} \\ \vdots \\ A\ \mathsf{mobile}\end{matrix}}{\mu\alpha.A\ \mathsf{mobile}}$$

$$\frac{}{\mathrm{w}\,\mathtt{addr}\ \mathsf{mobile}} \qquad \frac{A\ \mathsf{mobile} \quad B\ \mathsf{mobile}}{A \times B\ \mathsf{mobile}} \qquad \frac{A\ \mathsf{mobile} \quad B\ \mathsf{mobile}}{A + B\ \mathsf{mobile}}$$

$$\frac{A\ \mathsf{mobile}}{\forall\alpha.A\ \mathsf{mobile}} \qquad \frac{A\ \mathsf{mobile}}{\forall\omega.A\ \mathsf{mobile}}$$

Figure 2. Definition of the mobile judgment. Not all types are mobile: local resources like arrays and file descriptors are not, nor are function types or abstract types.

This program asserts the existence of a world `server` with a function that returns its version. On the home world, we display an alert whose argument is a subexpression (the call to `version`) that is evaluated at the server. The typing rule for `get` appears in Figure 1; it takes the address of a remote world and an expression well-typed at that world. The type of the expression must be mobile (Figure 2). A type is mobile if *all* values of that type are portable among worlds. `string` is mobile, so this code is well-typed. Function types are not mobile, because for example it would not make sense to move the function `version`—a resource local to `server`—to the world `home`. Even though not *all* functions are mobile, we will be able to demonstrate the mobility of particular functions with the ☘ modality, which is discussed in the next section.

3.1 Validity

It turns out that a large fraction of the code and data in a distributed application is not particular to any one world. We say that such values are "valid" and introduce a new kind of hypothesis for valid values. It takes the form $u{\sim}\omega.A$, meaning that the variable u is bound to a valid value which has type A at any world. The world variable ω is bound within A and is instantiated with the world(s) at which u is used. This variable is rarely needed, so we write $u{\sim}A$ when it does not occur in A. Valid hypotheses appear in Γ like the other hypotheses.

One way to introduce a valid hypothesis is with the `put` declaration. The code `put x = 2 + 3` binds a valid variable $x{\sim}$`int`. The typing rule for `put` appears in Figure 1; it requires that the type of the expression be mobile. Unlike `get`, `put` does not cause any communication to occur; it simply binds the variable such that it can be used in any world. There can also be global resources that are known to be available at all worlds. For example, `extern val server ~ server addr` declares that the address of the server is globally available.

The ☘ modality (pronounced "shamrock") is the internalization of the validity judgment as a type. A value of type ☘A is a value that can be used at any world at type A. It is introduced by checking that a value is well-typed at a hypothetical world about which nothing else is known (Figure 1). This hypothetical world can in general appear in the type; when it does, we write ☘$_\omega A$. Elimination of the ☘ modality with the `valid` declaration produces a valid hypothesis. Our treatment of validity and the ☘ modality are inspired by Park's $\bigcirc$ modality [12]. Note that the body of a `sham` (and also `hold`) must be a value. ML5 has several constructs that are syntactically restricted to values; in these positions we would not be able to safely evaluate an expression because it is typed at some other world.

The ☘ modality is useful because it analyzes a *particular value* for portability (compare the mobile judgment, which is judgment on types). Therefore it can be used to demonstrate the portability of a function value. For example, the ML5 expression `sham (fn x => x + 1)` has type `{}(int -> int)`. (`{}` is the ASCII syntax for ☘.) Because ☘A is mobile for any A, we can now `get` this wrapped function or place it inside other mobile data structures.

The programmer does not usually need to use the ☘ modality manually, because type inference will automatically generalize declarations to be valid when possible. This is described in the next section.

3.2 Polymorphism and Type Inference

Like Standard ML, ML5 supports Hindley-Milner style type inference. When the right hand side of a `val` declaration is a value (this includes any `fun` declaration), the compiler will generalize the free type variables to produce a polymorphic binding. ML5 also infers and generalizes worlds in the same manner. For instance,

```
val f = fn a => from a get 1234
```

produces a polymorphic binding of `f` at type $\forall\omega.(\omega$ `addr` $\supset$ `int`$)$. In ML5 world variables also appear in the judgment (the `@w` part), which is not in scope of the $\forall$ type operator and so cannot be generalized this way. If this world is unconstrained, the declaration is generalized to produce a valid binding by introducing and immediately eliminating the ☘ modality. The above code elaborates into

```
valid f = sham (allw w. fn (a : w addr) => from a get 1234)
```

which binds $f \sim \forall\omega.(\omega\,\texttt{addr} \supset \texttt{int})$.

Validity inference allows declarations of library code (such as the ML Basis library) to precede the program and then be used as desired, without the need to explicitly move the code between worlds or instantiate it. Thus, when programming using only the ML subset, ML5 looks just like ML.

3.3 Web Features

The current ML5 prototype is specialized to web programming, and has a few features that are specific to this domain. Let us look at a tiny application that illustrates these. This program will make use of a simple persistent database on the server that associates string values with string keys. It will allow us to modify those keys, and will asynchronously show the value of the key whenever it is modified (in this or any other session). We begin by importing libraries and declaring the worlds and addresses.

```
import "std.mlh"
import "dom.mlh"
import "trivialdb.mlh"
extern bytecode world server
extern val home ~ home addr
extern val server ~ server addr
```

The Document Object Model (DOM) is JavaScript's interface to the web page [5]. It allows the reading and setting of properties of the page's elements, and the creation of new elements. The `dom.mlh` library provides a simple interface to the DOM. The `trivialdb.mlh` library provides access to the persistent database. Both consist mainly of `extern` declarations.

```
put k = [tdb-test]
fun getkey () =
    let val v = from server get trivialdb.read k
    in  dom.setstring (dom.getbyid [showbox], [innerHTML], v) end
fun setkey () =
    let put s = dom.getstring (dom.getbyid [inbox], [value])
    in  from server get trivialdb.update (k, s) end
```

The valid variable `k` holds the name of the key we're concerned with (ML5 string constants are written with square brackets; see below). The function `getkey` fetches the current value of the key from the server. It then finds the DOM element with id *showbox* and sets its HTML contents to be the value of the key. The function `setkey` reads the value of the DOM element *inbox* (a text input box), travels to the server and sets that as the value of the key. Both functions have type `unit -> unit @ home`.

```
do dom.setstring (dom.getbyid [page], [innerHTML],
                  [[k]'s value: <div id="showbox">&nbsp;</div> <br />
                   <input type="text" id="inbox" /> <br />
                   <div onclick="[say setkey ()]"
                        style="cursor:pointer">set key</div> ])
do from server
   get trivialdb.addhook (k, cont (fn () => from home get getkey ()))
```

We then create the web page that the functions above interact with. We do this by updating the element called *page* (provided by the ML5 runtime) with an HTML string. This string contains the elements *showbox* and *inbox* referenced by name above. There are two things to note here: One is ML5's syntax for strings, which uses square brackets. Within a string, square brackets allow an ML5 expression of type `string` to be embedded (it may contain further strings, etc.). The other is the `say` keyword. It takes an ML5 expression (here `setkey ()`) and, at runtime, returns a JavaScript expression (as a string) that when run will evaluate that expression.[2] In this example we set the `onclick` property of the `<div>` so that it triggers `setkey ()` when the user clicks it. Finally, we add a hook on the key that travels to the client and calls `getkey` whenever the key is changed. The hook is expected to be a first-class continuation; `cont` is a valid function from the standard library of type `(unit -> unit) -> unit cont`.

When this program is compiled, it produces a JavaScript source file to run on the client, and a bytecode file to run on the server. To run the application, the user visits a URL on the web server, which creates a new session and returns the JavaScript code along with the runtime to his web browser. The server also launches an instance of its code. The program runs until the client leaves the web page, at which point the session is destroyed on the server. This example and others can be run online at http://tom7.org/ml5/.

Having given a tour of the language, we now describe how it is implemented.

4 Implementation

The ML5 implementation consists of a CPS-based type directed compiler, a simple web server, and two runtimes: one for the client and one for the server. For reasons of space we concentrate on only the most interesting aspects of these, which tend to arise as a result of ML5's modal typing judgment.

We first discuss our strategy for marshaling, which pervasively affects the way we compile. We then discuss the phases of compilation in the same order they occur in the compiler. We finish with a brief discussion of the runtime system.

Marshaling. The design of ML5 maintains a conceptual separation between marshaling and mobility. Marshaling is an implementation technique used to represent values in a form suitable for transmission over the network. Mobility is a semantic quality of values determined at the language level. In ML5, any well-typed value can be marshaled, but only some values are mobile. We are able to make this distinction because of the modal typing judgment: when a value of type $A\,@\,\mathrm{w}_1$ is marshaled and then unmarshaled at w_2, it still has type $A\,@\,\mathrm{w}_1$ and therefore cannot be consumed at w_2. The notion of mobility allows us to coerce some values of type $A\,@\,\mathrm{w}_1$ to $A\,@\,\mathrm{w}_2$.

[2] We can not provide any type guarantees about JavaScript once it is in string form. An improvement would be to use a richer language for embedded XML documents (like Links; see Section 6) so that we can type check them, and then to have `say` return a JavaScript function object rather than a string.

$$\begin{array}{rl} \text{values } v ::= & x \mid u \mid \texttt{sham}\ \omega.v \mid \lambda x.c \mid v_1\langle \overline{\mathrm{w}}; \overline{A}; \overline{v}\rangle \mid \Lambda\langle\overline{\omega};\overline{\alpha};\overline{x{:}A}\rangle.v \mid \\ & \texttt{wrepfor}\ \mathrm{w} \mid \texttt{repfor}\ A \end{array}$$

$$\text{conts } c ::= \texttt{halt} \mid \texttt{go}[v]c \mid \texttt{letsham}\ u = v\ \texttt{in}\ c \mid \texttt{leta}\ x = v\ \texttt{in}\ c \mid \texttt{call}\ v_f\ v_a$$

$$\dfrac{}{\Gamma \vdash \texttt{wrepfor}\ \mathrm{w} : \mathrm{w}\ \texttt{wrep} @ \mathrm{w}'} \qquad \dfrac{\Gamma, \overline{\omega}, \overline{\alpha}, \overline{x{:}A} \vdash v : B @ \mathrm{w}}{\Gamma \vdash \Lambda\langle\overline{\omega};\overline{\alpha};\overline{x{:}A}\rangle.v : \langle\overline{\omega};\overline{\alpha};\overline{A}\rangle.B @ \mathrm{w}}$$

$$\dfrac{}{\Gamma \vdash \texttt{repfor}\ A : A\ \texttt{rep} @ \mathrm{w}} \qquad \dfrac{\Gamma \vdash v_f : \langle\overline{\omega};\overline{\alpha};\overline{A}\rangle.B @ \mathrm{w}_0 \quad \overline{\Gamma \vdash v : A @ \mathrm{w}_0}}{\Gamma \vdash v_f\langle\overline{\mathrm{w}};\overline{C};\overline{v}\rangle : [\overline{\mathrm{w}}/\overline{\omega}][\overline{C}/\overline{\alpha}]B @ \mathrm{w}_0}$$

$$\dfrac{\Gamma \vdash v : A @ \mathrm{w}'}{\Gamma \vdash \texttt{hold}\ v : A\ \mathsf{at}\ \mathrm{w}' @ \mathrm{w}} \qquad \dfrac{}{\Gamma, x{:}A @ \mathrm{w}, \Gamma' \vdash x : A @ \mathrm{w}} \qquad \dfrac{}{\Gamma, u{\sim}\omega.A, \Gamma' \vdash u : [\mathrm{w}/\omega]A @ \mathrm{w}}$$

$$\dfrac{\Gamma \vdash v_a : \mathrm{w}'\ \mathsf{addr} @ \mathrm{w} \quad \Gamma \vdash c @ \mathrm{w}'}{\Gamma \vdash \texttt{go}[v_a]c @ \mathrm{w}} \qquad \dfrac{A\ \mathsf{cmobile} \quad \Gamma \vdash v : A @ \mathrm{w} \quad \Gamma, u{\sim}A \vdash c @ \mathrm{w}}{\Gamma \vdash \texttt{put}\ u = v\ \texttt{in}\ c @ \mathrm{w}}$$

$$\dfrac{\Gamma \vdash v : \text{☘}_\omega A @ \mathrm{w} \quad \Gamma, u{\sim}\omega.A \vdash c @ \mathrm{w}}{\Gamma \vdash \texttt{letsham}\ u = v\ \texttt{in}\ c @ \mathrm{w}} \qquad \dfrac{\Gamma \vdash v : A\ \mathsf{at}\ \mathrm{w}' @ \mathrm{w} \quad \Gamma, x{:}A @ \mathrm{w}' \vdash c @ \mathrm{w}}{\Gamma \vdash \texttt{leta}\ x = v\ \texttt{in}\ c @ \mathrm{w}}$$

Figure 3. Some of the CPS language. The judgment $\Gamma \vdash v : A @ \mathrm{w}$ checks that the value v has type A at w. Continuation expressions c are checked with the judgment $\Gamma \vdash e @ \mathrm{w}$; they do not return and so do not have any type. The judgment $\mathsf{cmobile}$ is analogous to the IL judgment mobile, but for CPS types. In an abuse of notation, we use an overbar to indicate a vector of values, vector of typing judgments, or simultaneous substitutions.

In order to marshal and unmarshal values, we need dynamic information about their types and worlds. For example, to compile a polymorphic function, we might need to generate code that marshals a value of an arbitrary type. To do this uniformly, the low-level `marshal` primitive takes a value (of type A) and a representation of its type. The type of the dynamic representation of A is $A\,\texttt{rep}$. (We also have $\mathrm{w}\,\texttt{wrep}$, the type of a representation of the world w.) The `marshal` primitive analyzes the type representation in order to know how to create marshaled bytes from the value. Recursively, the behavior of marshal is guided by both the type and world of the value. Because `marshal` is a primitive—not user-defined code—we do not need to support general type analysis constructs like typecase.

To make sure that we have the appropriate type representation available when we invoke `marshal`, we establish an invariant in the compiler that whenever a type variable α is in scope, so is a valid variable with type $\alpha\,\texttt{rep}$. Similarly, for every world variable ω in scope, there will be a valid variable with type $\omega\,\texttt{wrep}$. Once we have generated all of the uses of these representations, we discard the invariant and can optimize away any type representations that are not needed.

4.1 Compiler

After the program is elaborated into the intermediate language (IL), the first step is to CPS convert it. CPS conversion is central to the implementation of threads and tail recursion in JavaScript, because JavaScript does not have any native thread support or tail call support, and has an extremely limited call stack. We give a sample of the CPS language in Figure 3. CPS conversion of most constructs is standard [1]; IL expressions are converted to CPS expressions via a function `convert`, which is itself continuation-based. In addition to the IL expression argument, it takes a (meta-level) function $\mathcal{K}$ that produces a CPS expression from a CPS value (the result value of M). It may be illuminating to see the case for `get`:

```
convert (from $M_a$ get $M_r$) $\mathcal{K}$ =
        convert $M_a$ $\mathcal{K}_1$
          where $\mathcal{K}_1(v_{a'})$ = let $a$ = localhost() in
                                put $u_a = a$ in
                                go[$v_{a'}$] convert $M_r$ $\mathcal{K}_r$
          where $\mathcal{K}_r(v_r)$ =  put $u_r = v_r$ in
                                go[$u_a$] $\mathcal{K}(u_r)$
```

We first convert M_a, the address of the destination, and then compute our own address and make it valid so that we can use it at our destination to return. We then `go` to the destination, evaluate the body M_r, and make it valid so that we can use it when we return. To return, we `go` back to the original world.

The CPS abstract syntax is implemented in the compiler using a "wizard" interface [8], where binding and substitution are implemented behind an abstraction boundary rather than exposing a concrete SML datatype and relying on compiler passes to respect its binding structure. This interface guarantees that every time a binder is "opened," the client code sees a new freshly alpha-varied variable. In our experience this is successful in eliminating alpha-conversion bugs, a major source of mistakes in compilers we have written previously.

Because the compiler is type-directed, all of the transformations are defined over typing derivations rather than the raw syntax. In order to recover these derivations (particularly, the types of bound variables) each transformation must essentially also be a type checker. We do not want to repeat the code to type check and rebuild every construct in every pass. Instead, we define an identity pass that uses open recursion, and then write each transformation by supplying only the cases that it actually cares about. This does have some drawbacks (for instance, we lose some useful exhaustiveness checking usually performed by the SML compiler), but it drastically reduces the amount of duplicated code that must be maintained in parallel.

Representation Insertion. The first such pass establishes the representation invariant mentioned above. A number of constructs must be transformed: constructs that bind type or world variables must be augmented to additionally take representations, and uses must provide them. The CPS language uses a "fat lambda" (written $\Lambda\langle\overline{\omega};\overline{\alpha};\overline{x{:}A}\rangle.v$) for values that take world, type, and value arguments. It is converted by adding an additional value argument (of type $\omega\,\texttt{wrep}$ or $\alpha\,\texttt{rep}$) for each world and type argument. As examples, the value $\Lambda\langle\omega_1,\omega_2;\alpha;x{:}\texttt{int}\rangle.x$ converts to

$$\Lambda\langle\omega_1,\omega_2;\alpha;x_1{:}\omega_1\,\texttt{wrep},x_2{:}\omega_2\,\texttt{wrep},x_3{:}\alpha\,\texttt{rep},x{:}\texttt{int}\rangle.x$$

and the application $y\langle\textbf{home},\omega_3;(\texttt{int}\times\alpha);0\rangle$ converts to

$$y\langle\textbf{home},\omega_3;(\texttt{int}\times\alpha);\texttt{wrepfor}\ \textbf{home},\texttt{wrepfor}\ \omega_3,\texttt{repfor}\ (\texttt{int}\times\alpha),0\rangle.$$

The value $\texttt{repfor}\ A$ is a placeholder for the representation of A. It is only a placeholder because it may contain free type variables. In a later phase, `repfor` is replaced with a concrete representation, and the free type variables become free valid variables.

We also perform a similar translation for the $\texttt{sham}\ \omega.v$ and `letsham` constructs. For the introduction form, we lambda-abstract the required world representation. We do not change the elimination site, which binds a valid variable that can be used at many different worlds. Instead, at each use we apply the variable to the representation of the world at which it is used.

In this phase we also insist that every `extern type` declaration is accompanied by a `extern val` declaration for a valid representation of that type.

Closure Conversion. Closure conversion implements higher-order, nested functions by transforming them to closed functions that take an explicit environment. The environment must contain all of the free variables of the function. Closure conversion is interesting in the ML5 implementation because a function may have free variables typed at several different worlds, or that are valid.

To closure convert a lambda, we compute the free variables of its typing derivation. This consists of world, type, and value variables. After closure conversion the lambda must be closed to all dynamic variables, including the representations of types and worlds. This means that in order to maintain our type representation invariant, the set of free variables must additionally include a valid representation variable for any occurring world or type variable. So the free variables $x_i{:}A_i\,@\,\mathrm{w}_i$ are the actually occurring free variables, and the free valid variables are $u_j{\sim}\omega.B_j$ along with $u_{\omega_k}{\sim}\omega_k\ \texttt{wrep}$ for any free ω_k and $u_{\alpha_l}{\sim}\alpha_l\ \texttt{rep}$ for any free α_l, where u_ω is the representation variable paired with a world variable ω and u_α is the variable pared with a type variable α.

The environment will consist of a tuple containing the values of all of the free variables. Some of these values are typed at other worlds, so they need to be encapsulated with the `at` modality. We must preserve the validity of the valid ones using $\boxtimes$. The environment and its type are thus

$$(\texttt{hold}\ x_1,\ \ldots,\ \texttt{sham}\ \omega.u_1,\ \ldots) : (A_1\,\texttt{at}\,\mathrm{w}_1,\ \ldots,\ \boxtimes_\omega.B_1,\ \ldots)$$

Inside the body of the converted function we rebind these variables using `leta` and `letsham` on components of the tuple. As usual, the pair of the closed lambda and its environment are packed into an existential, so that all function types are converted independently of the instance's free variable set. Since unpacking an existential binds a type variable, we must also include a type representation inside each existential package so that we can maintain our invariant.

The design of closure conversion is what originally informed our addition of the `at` and ☘ modalities to ML5. A general lesson can be derived: In order to type closure conversion, the language must have constructs to internalize as types any judgments that appear as (dynamic) hypotheses. The elimination forms must be able to restore these hypotheses from the internalized values.

In addition to closure converting the regular λ construct, we must convert Λ since it takes value arguments. We closure convert the body of `go` as well, since we send that continuation as a marshaled value to the remote host.

After closure conversion we will never need to insert another `repfor`, so a pass replaces these with the actual values that form the runtime representations of types and worlds. We then discard our representation invariant and can optimize away unused representations.

Hoisting. A separate process of hoisting pulls closed lambdas out of the program and assigns them global labels. The hoisted code must be abstracted over all of its free type and world variables, but these are now purely static. Hoisted code can either be fixed to a specific world (when it is typed at a world constant), or it can be generic in its world (when it is typed at a world variable). When we generate code for each world in the back-end, we produce code for those labels that are known to be at that world, and also any label generic in its world. Any other label is omitted—it won't be invoked here and we might not even be able to generate the code if it makes use of resources particular to its true world.

4.2 Runtime

The runtime system is responsible for providing communication between the server and client. It also contains the implementation of marshaling and threads.

When the web server returns a compiled ML5 program for the client, it begins a session of the program that runs on the server as well. This session contains a queue of server threads and a marshaling table (see below). Via the `go` construct, threads can transfer control from client to server or vice versa. A client thread transfers control to the server by making an HTTP request whose body is a marshaled continuation for the server to execute. Starting a thread on the client is trickier: For security reasons JavaScript cannot accept incoming network connections. Instead, the client is responsible for maintaining a devoted connection to the server, fetching a URL and asynchronously waiting on that request. When the server wishes to start a thread on the client, it sends a response; the client begins that thread and reinstates the connection. (This mode of interaction is now fairly standard in web applications.)

With type representations available, marshaling is a straightforward matter. One interesting aspect is how we use the representations of worlds; as we marshal, we recursively keep track of the world of the value (for instance, as we descend into a value of type A `at` w_2, we record w_2). We can then specialize the marshaled representation of a value based on where it comes from. This is how we can marshal local resources: A JavaScript DOM handle is represented natively at `home`, but when we marshal it, we place it into a local table and marshal the index into that table. At any other world, the handle is represented and marshaled as this index. When it arrives back at `home`, we know to reconstitute the actual handle by looking it up in the table.

5 Theory

We have formalized several of the calculi on which ML5 is based in Twelf [14] and proved properties such as type safety for them. In addition, we have formalized a few of the first stages of compilation, including CPS and closure conversion. (These languages are somewhat simplified; for example we omit recursion and type representations.) For these we prove that every well-typed program can be converted, and that the resulting program is well-typed. All of the proofs are machine checkable. Some of the proofs appear in Murphy's thesis proposal [9] and the remainder will appear in his dissertation.

6 Related and Future Work

Related Work. ML5 is in a class of new web programming languages that Wadler deems "tierless," that is, they allow the development of applications that normally span several tiers (database, server logic, client scripts) in a uniform language. Links [2] is such a programming language. Functions may be annotated as "client" or "server," and Links allows calls between client and server code. However, their type system does no more to indicate what code and data can be safely mobile, and marshaling can fail at runtime. On the other hand, Links has many features (such as a facility for embedding and checking XML documents and database queries) that make typeful web programming easier.

Hop [15] is another tierless web programming language, using Scheme as the unifying language. Hop has constructs for embedding a client side expression within a server expression and vice-versa, analogous to `get` in ML5 (but specific to the two-world case). The chief difference is simply that Hop is untyped, and thus subject to dynamic failures.

Others have used modal logic for distributed computing. We have already mentioned a few of these; for a complete discussion see our previous papers on Lambda 5 [11] and C5 [10], as well as Murphy's thesis proposal [9].

Future Work. There is much potential for future work on ML5 and related languages. In the short term, we wish to develop larger applications and implement the language support necessary to do so. This will probably include support for structured databases and mutual exclusion between threads. We will need to improve the performance of the compiler, particularly by implementing optimizations that undo our simplistic closure conversion (for instance, when all calls are direct) and type representation passing (when the representations are not used). There is also some opportunity for optimizations particular to the ML5 primitives (such as when a `get` is from a world to itself).

A more serious performance issue is resource leaks caused by mobile data structures. Garbage that is purely local is collected by the server and JavaScript collectors, but once a local resource is marshaled by inserting it in a table and sending that index remotely, we can never reclaim it. Web programming systems typically deal with this by assuming that sessions are short-lived, but it would be preferable to allow for long-running programs through some form of distributed garbage collection [13].

Our type theory naturally supports an arbitrary number of worlds, and most of the compiler does, as well. Adding the ability for a program to access many different servers would just be a matter of adding runtime support for it. Unfortunately, JavaScript's security model prevents outgoing connections to any server other than the one that originally sent the JavaScript code. To get around this, we would need to build an overlay network where the original server acts as a proxy for the others. Supporting multiple *clients* in the same application instance is trickier still. This is mainly because we consider the thread of control to begin on the (one) client; it would instead need to start on the server, which would then need to be able to discover new connecting clients at runtime.

Another concern is security. JavaScript code intended to run on the client is actually under the complete control of an attacker. He can inspect its source and cause it to behave arbitrarily, and invoke any continuation on the server for which he is able to craft acceptable arguments. This is true of any distributed system where some hosts are controlled by attackers, and the programmer must defend against this by not trusting (and explicitly checking) data and code it receives from the client. In some ways this problem is exacerbated in ML5: The process of compilation from the high-level language is not fully abstract, in that it introduces the possibility for more behaviors in the presence of an attacker than can be explained at the source level. For example, depending on how closure conversion and optimizations are performed, a client may be able to modify a marshaled closure in order to swap the values of two of the server's variables! We believe a solution to this problem would take the form of an "attack semantics" provided by the language and implemented by the compiler through a series of countermeasures. The semantics would describe the range of behaviors that a program might have in the presence of an attacker, so that the programmer can ensure that these behaviors do not include security breaches on the server. (The client will always be able to format his own hard drive, if he desires.) Such properties are inherently in terms of the principals (places) involved in the computation, and therefore we believe that our type system and semantics is an important first step in being able express and prove properties of programs in the presence of an attacker, and to develop mechanisms for building secure programs.

Conclusion. We have presented ML5, a new programming language for distributed computing. ML5's current prototype is specialized to web programming, a domain for which its programming model is well suited—it joins a collection of other languages with similar design goals and principles. Many of the ideas from these languages are compatible with all three systems. ML5's main contribution to this is its type system, which permits the programmer to describe local resources and prevent unsafe access to them. Being based on logic, the type system is elegant and is compatible with the design of ML-like languages, including polymorphic type inference.

References

1. Andrew Appel. *Compiling With Continuations*. Cambridge University Press, Cambridge, 1992.
2. Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *5th International Symposium on Formal Methods for Components and Objects (FMCO)*. Springer-Verlag, 2006. To appear.
3. ECMAScript language specification. Technical Report ECMA-262, 1999.
4. Hybrid logics bibliography, 2005. URL: http://hylo.loria.fr/content/papers.php.
5. W3C DOM IG. Document object model, 2005. http://w3c.org/DOM/.
6. Limin Jia and David Walker. Modal proofs as distributed programs (extended abstract). *European Symposium on Programming*, 2004.
7. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997.
8. Tom Murphy, VII. The wizard of TILT: Efficient(?), convenient and abstract type representations. Technical Report CMU-CS-02-120, Carnegie Mellon School of Computer Science, 2002.
9. Tom Murphy, VII. Modal types for mobile code (thesis proposal). Technical Report CMU-CS-06-112, Carnegie Mellon, Pittsburgh, Pennsylvania, USA, 2006.
10. Tom Murphy, VII, Karl Crary, and Robert Harper. Distributed control flow with classical modal logic. In Luke Ong, editor, *14th Annual Conference of the European Association for Computer Science Logic (CSL 2005)*, Lecture Notes in Computer Science. Springer, August 2005.
11. Tom Murphy, VII, Karl Crary, Robert Harper, and Frank Pfenning. A symmetric modal lambda calculus for distributed computing. In *Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS 2004)*. IEEE Press, 2004.
12. Sungwoo Park. A modal language for the safety of mobile values. In *Fourth ASIAN Symposium on Programming Languages and Systems*, November 2006.
13. David Plainfossé and Marc Shapiro. A survey of distributed collection techniques. Technical report, BROADCAST, 1994.
14. Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In D. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, pages 120–135, Rome, Italy, September 2003. Springer-Verlag LNCS 2758.
15. M. Serrano, E. Gallesio, and F. Loitsch. HOP, a language for programming the Web 2.0. In *Proceedings of the First Dynamic Languages Symposium*, 2006.
16. Alex Simpson. *The Proof Theory and Semantics of Intuitionistic Modal Logic.* PhD thesis, University of Edinburgh, 1994.