Modal Types for Mobile Code (Thesis Proposal) [*]

Tom Murphy VII

March 13, 2006

Abstract

Modal logic is a family of logics with the ability to simultaneously reason about truth from multiple perspectives. Our previous work showed how the modal logic Intuitionistic S5 could form the basis for a simple lambda calculus for spatially distributed programs. I propose here a thesis project to demonstrate the efficacy and elegance of modal type systems for controlling spatially distributed resources in a programming language. The project has strong components of both theory and practice: the design of ML5, a new programming language for distributed computing, and its implementation. I present technical details for the language and implementation where they have been completed, and a plan for the work to be done where they have not.

1 Introduction

This research seeks to demonstrate that modal type systems provide an elegant and practical way to control spatially distributed resources in mobile computer programs. I propose to do this by designing and implementing a programming language with a modal type system. I will then demonstrate the language and implementation's effectiveness by building an example distributed application in the language.

The work is part of a larger research program, called ConCert, which is building a large-scale peer-to-peer "Grid Computing" platform based on certified code technology. For this proposal, I will begin by describing the current design of ConCert, which will serve to motivate the thesis work and argue for my qualifications to undertake the proposed research. I will then demonstrate how modal types are an appropriate tool for reasoning about spatial distribution, by reviewing a foundational calculus we developed [31, 30] based on modal logic. Given the problem and an appropriate tool, I then present the design for a new programming language for distributed computing called ML5.

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

Before going into these items in detail, a high-level summary of each point should serve to provide the necessary context to understand what follows.

1.1 Overview

ConCert. The ConCert project's goal is to build a large-scale peer-to-peer distributed network for trustless Grid computing [11]. Grid computing [18], which is named by analogy with electric power grids, is the practice of coupling diverse computational resources into a large shared computer. In this setting, code moves around the network, which presents a security issue: How does a code consumer know that a piece of code is safe to run? There are several functioning Grid computing systems already deployed [17, 3], and each relies on the establishment of trust relationships between the code producer and code consumer to address this security issue. In the ConCert system, we allow for the *trustless* dissemination of code by using certified code technology [32]. To achieve this, each piece of mobile code comes with a certificate demonstrating the relevant properties of the code. These are intrinsic properties of the code (*e.g.*, its type safety) rather than the extrinsic properties (*e.g.*, who wrote it) sometimes certified by cryptographic signatures.

The implementation of the proposed language, ML5, should also work by producing safety certificates. I will do this by building a type-directed certifying compiler [27, 35]. This means that the implementation must be carefully designed (see Section 6), but doing so has many research and engineering benefits.

Grid/ML. Grid/ML [29] is a high-level language that can be used to produce certified code for use on the ConCert network. It is a functional language based on Standard ML [24] with primitives for Grid computing. In Grid/ML, the programmer writes his whole Grid application as a single program. This program is made of two parts: the client, which runs only on the user's computer and interacts with him, and the mobile code, which runs on arbitrary hosts in the network.

Grid/ML is practical for a certain class of problems, and has been used for a few Grid programming experiments [12]. However, it has two major shortcomings. First, the mobile code is agnostic to where it is running, and so applications must treat the hosts in the network uniformly; it is not possible to make use of special resources only available at certain sites. Since distributed computing is often motivated by the desire to make use of resources other than mere processing power, this rules out an important set of applications.

Second, even with the assumption of uniformity, Grid/ML still has a distinction between client code and mobile code. Certain operations, such as I/O, can only be performed on the client and result in run-time failure if used in mobile code.

My solution to both problems is to enrich the ML type system with a concept of *place*. By doing so, the language will be able to support location-aware programming, so that network locations need not be treated uniformly. As a consequence, the client will be a place like any other, and we will be able to distinguish its capabilities from the capabilities of other hosts. Because we use a type system, we can make these distinctions statically, excluding errors before the program is ever run.

Modal logic and modal types. Type systems for functional languages like ML have a close connection to logic, called the Curry-Howard isomorphism. Under this view, the propositions of intuitionistic logic are interpreted as the types of a programming language. Proofs of propositions then become programs inhabiting those types. Different logics, viewed through the lens of the Curry-Howard isomorphism, give rise to a variety of elegant and useful type systems for programming languages.

In order to develop a type system with a notion of place, we use a logic with the ability to reason spatially, namely modal logic [23]. The propositional logic upon which ML is based is concerned with the *truth* of propositions from a single universal viewpoint. Modal logic introduces the concept of truth from multiple different perspectives, which are called "worlds." The logic is then able to reason simultaneously about truth in these worlds. Under the Curry-Howard view, these worlds become hosts in the network, and so our type system is correspondingly endowed with a notion of place.

Distributed programming with modal types. Because a proof in modal logic contains reasoning from multiple different worlds, programs in our modal programming language span multiple hosts. That is, each program consists of a series of nested expressions to be evaluated at various hosts in the network. These fragments make reference to the other sites in the network, and the resources particular to those sites. The main purpose of the type system is to track these references to remote resources so that they are only used in a safe way: Localized resources can only be used in the correct place. When code or data do not depend on their location, we are also able to indicate this in the type system. This allows them to be safely used anywhere. The main result of type safety of the language is that resources are only used in the correct place. This brings us to the thesis statement:

> **Thesis Statement.** Modal type systems provide an elegant and practical way to control spatially distributed resources in mobile computer programs.

Although modal types naturally address spatial distribution, this is only one facet of distributed computing. In particular, a logical approach to concurrency and failure is beyond the scope of this thesis. However, the language will need to have basic support for these features in order to create a realistic application.

Implementation. The implementation of ML5 will consist of a type-directed compiler for the language and a distributed runtime system. The compiler is responsible for translating the high-level language into certified machine code. The runtime's primary task is to form the network of hosts on which the distributed programs run. To do this, it must be able to marshal code and data into raw bytes so that they can be transmitted on the network. In order to ensure safety, the marshalled representation must be accompanied by a certificate demonstrating its well-formedness. It is the runtime's responsibility to verify these certificates upon receipt, relative to the host's specific set of local resources. As the marshalled bytes are reconstituted into code and data, references to those local resources become linked to them.

The runtime has a few other responsibilities: it performs distributed garbage collection to reclaim resources, and provides rudimentary support for failure detection and concurrency (Section 7).

Application. To demonstrate the usefulness of the language and implementation, I will use it to build an example distributed application. Before choosing an appropriate application, I will need to have a better idea of how the language supports features like concurrency. Thus, this proposal does not commit to a specific application.

This concludes the high-level summary. The remainder proceeds as follows: I begin by explaining the proposed work in depth using the same outline as above (Sections 2–8). For the language design, I give both informal code examples and formal semantics for a simplified subset of the language. For the implementation of the compiler, I give formal translations for the core language constructs; these translations need only be extended to support a more full-featured language. For the compiler's back-end and runtime, I outline the research problems that still need to be solved. I conclude with a comparison to other languages for distributed computing (Section 9), and a timeline for the completion of the thesis work (Section 10).

2 ConCert

The current ConCert system is designed to harness one specific resource: idle CPU power from a large network of volunteered computers. A program designed to be run on ConCert is broken up into pieces of independent mobile code (called "cords") that may run in parallel. To the programmer, ConCert acts as a single, highly parallel computer with simple primitives for fork-join parallelism (see below).

A computer becomes part of the ConCert network by running a piece of software called the Conductor. The Conductor is responsible for maintaining contact with its peers, and for allocating cords to idle computers in order to be executed. This allocation is done using a "work-stealing" model: Each host maintains a queue of cords that are ready to be executed, and when a host is idle, it "steals" work out of the queues of other hosts. This model is efficient [5], but because the location in which a cord will eventually run depends on the idle status of the machines in the network, it is not possible for the programmer to know in advance where his code will run. Moreover, to support fault tolerance (by restarting failed cords), a cord must be able to be run multiple times in different places and always produce the same result.

As a consequence, we provide cords with a uniform view of the network—a cord cannot tell what host it is running on, nor can it access any of that host's local resources like permanent storage and I/O. This is acceptable because the only resource ConCert seeks to harness is idle CPU power. In contrast, the proposed work will allow programs to make use of any sort of distributed resource.

Adding language support for such local resources is tricky. To illustrate some of the issues, we look at a simple Grid/ML program.

```
let
   val f  = openfile "numbers.txt"
   val f2 = openfile "factors.txt"

   (* append the list l to the output file *)
   fun writeresult l =
      write(f2, nums-to-string l ^ "\n")

   val inputs = readfile f

   (* prime factorization of n for n > 0 *)
   fun factor n =
      let
        fun trial 1 = n :: nil
          | trial m = if n mod m = 0
                      then factor m @ factor (n div m)
                      else trial (m - 1)
      in
        trial (floor (sqrt (real n)))
      end

   val cords =
     map (fn n => submit (fn () => factor n)) inputs

   val results = waitall cords
in
   app writeresult results
end
```

This Grid/ML program[1] computes the prime factorizations of the numbers

[1]The example takes the liberty of assuming the existence of some library code, for instance

in the file `"numbers.txt"` and writes those to the file `"factors.txt"` (both files reside on the client). It does this by creating a cord for each factoring task and submitting them to the local work queue with the primitive `submit`. The Grid/ML primitives used have these types:

```
submit : (unit → α) → α cord
waitall : α cord list → α list
```

A value of type α `cord` is a running computation that returns a result of type α. Grid/ML allows α to be instantiated with any type. In this case each job is an `int list cord`. After submitting the cords, the program then waits for them all to complete, and writes the results to the output file.

The file I/O that the program performs is an effect. Such effects are allowed only in the part of the Grid/ML program that runs on the client. This is because I/O interacts with the external world, and so it depends on the place in which the code is executed. Therefore, an effect inside cord code would violate the requirement that it be agnostic about the host on which it runs. In Grid/ML, such errors are detected only at run-time. For example, if we modify the program so that the body of `factor` tries to write to the file `f2` or read from the file `f`, then the program will abort at that operation. Although both files are in scope, the file descriptors do not make sense when executing at remote sites—even if we wanted to allow I/O in cord code, we would not be able to access *those* files once execution has left the client.

This does not necessarily mean that open file descriptors cannot ever leave the client. Suppose the program is modified to be higher-order:

```
let
   (* ... *)

   fun factor n =
      let
        (* ... *)
        val factors = trial (floor (sqrt (real n)))
      in
        (fn () => writeresult factors)
      end

   (* ... *)
in
   app (fn g => g ()) results
end
```

Now, instead of returning the list of factors directly, each cord returns a function that writes the result to the file on the client. The client consumes these results by calling the functions. (Writing the program this way is gratuitous

`nums-to-string` and `readfile`.

here, but higher-order mobile code is often genuinely useful!) This program has the same behavior as the first version. The reference to the local file safely makes a round trip from the client to the cord code and back in the environment of each function. Thus it is not the references themselves that we need to tame, but the way those references to local resources are *used*. To do this, the type system of ML5 will associate a place with each value, and then only allow a value to be consumed in that same place. However, values will be able to flow freely between places otherwise.

Associating a place with each value does ensure safety, but it can exclude too many programs; many values are actually portable to every location. We do not want such values to be unnecessarily restricted to the place where they are created. For example, in the program above the `factor` function refers to the functions `sqrt` and `@`, which are defined as part of the standard library on the client. This is safe because these functions do not use any local resources. An important feature of the ML5 type system is that it also accounts for values like these that can be used anywhere. Using these features the ML5 type system will allow us to statically typecheck both the original program and the higher-order version.

We have now motivated a type system enhanced with a notion of place. To summarize, the type system for ML5 will ensure that localized resources are only used in the correct place by associating with each sub-expression the place in which it will be evaluated, and associating with each bound variable the place where it may be used. We will also introduce support for bindings that are usable in any place, since this is very common. The foundation of this type system comes from modal logic, which is introduced in the next section.

3 Modal Logic

"Modal logic" refers to a family of logics with the ability to reason about truth from multiple simultaneous perspectives. For this proposal, we will start with the specific modal logic called Intuitionistic S5 and then extend it to suit our purposes. IS5 is most easily understood as an extension of Intuitionistic logic, whose basic judgment is

$$\Gamma \vdash A \;\mathsf{true}$$

where A is a proposition and Γ is a collection of hypotheses of the form B true for various propositions B. The judgment asserts that the proposition A is true, from a single universal viewpoint, assuming that the hypotheses in Γ are also true.

IS5 changes this judgment by indexing the notion of truth by a collection of "possible worlds." These possible worlds are the different perspectives from which we judge the truth of propositions. All possible worlds must indeed be possible (they must not admit logical contradictions), but otherwise can disagree on what facts are true. As a real world example, in some possible worlds the proposition "it is raining in Pittsburgh" is true and in others it is not. However, in no possible world is "it is raining in Pittsburgh and it is not raining in Pittsburgh" true. Modal logic gives us the ability to state that a proposition is true in some specific possible world. The basic judgment in IS5 is:

$$\Gamma \vdash A\ \mathsf{true@}\mathrm{w}$$

where A is a proposition as before and w is a world. This judgment asserts that, assuming the hypotheses in Γ, the proposition A is true in the world w. World expressions w include both constants **w** standing for specific worlds, and world variables, written ω. Truth in IS5 is always with respect to a world. The context Γ therefore consists of assumptions of the form $B\ \mathsf{true@}\mathrm{w}'$.

Most of the rules from intuitionistic logic are imported into IS5, with the judgment simply changed from true to $\mathsf{true@}\mathrm{w}$. For example, we reason about conjunction as before:

$$\frac{\Gamma \vdash A\ \mathsf{true@}\mathrm{w} \quad \Gamma \vdash B\ \mathsf{true@}\mathrm{w}}{\Gamma \vdash A \wedge B\ \mathsf{true@}\mathrm{w}}\ {\scriptstyle \wedge \mathrm{I}} \qquad \frac{\Gamma \vdash A \wedge B\ \mathsf{true@}\mathrm{w}}{\Gamma \vdash A\ \mathsf{true@}\mathrm{w}}\ {\scriptstyle \wedge \mathrm{E}_1}$$

To conclude $A \wedge B$ at the world w, we must conclude both A and B at that same world. Likewise, if we know $A \wedge B$ at w, then we know A at that same world.

There are also two new connectives, which both use the indexed truth judgment in interesting ways. The first is $\Box$ (read "box"), where $\Box A$ means that A is true in all possible worlds. The rules for $\Box$ are:

$$\frac{\Gamma, \omega\ \mathsf{world} \vdash A\ \mathsf{true@}\omega}{\Gamma \vdash \Box A\ \mathsf{true@}\mathrm{w}}\ {\scriptstyle \Box \mathrm{I}} \qquad \frac{\Gamma \vdash \Box A\ \mathsf{true@}\mathrm{w}}{\Gamma \vdash A\ \mathsf{true@}\mathrm{w}'}\ {\scriptstyle \Box \mathrm{E}}$$

Because the meaning of $\Box A$ is that A is true in every world, we may conclude from $\Box A$ at some world w the proposition A at any world w' (rule $\Box$ E). To prove that a proposition is true in every world (rule $\Box$ I), we must prove that the proposition A is true at a new hypothetical world ω about which nothing is known. This world variable is bound in Γ.

The other new connective is $\Diamond$ (read "diamond"). The proposition $\Diamond A$ means that A is true in *some* possible world. The $\Diamond$ connective is defined as follows:

$$\frac{\Gamma \vdash A\ \mathsf{true@}\mathrm{w}'}{\Gamma \vdash \Diamond A\ \mathsf{true@}\mathrm{w}}\ {\scriptstyle \Diamond \mathrm{I}} \qquad \frac{\Gamma \vdash \Diamond A\ \mathsf{true@}\mathrm{w}' \quad \Gamma, \omega\ \mathsf{world}, A\ \mathsf{true@}\omega \vdash C\ \mathsf{true@}\mathrm{w}}{\Gamma \vdash C\ \mathsf{true@}\mathrm{w}}\ {\scriptstyle \Diamond \mathrm{E}}$$

To prove $\Diamond A$ at w, we require a proof of A in any world w' (rule $\Diamond$ I). To consume a proof of $\Diamond A$ (rule $\Diamond$ E), we get to assume the existence of some world (about which nothing else is known) where A is true, in order to continue proving some other proposition C. The elimination of $\Diamond A$ happens at a different world (w') than the remainder of the reasoning (w). In our computational interpretation (Section 4), these worlds will be thought of as hosts in the network, and so this rule incurs a sort of "action at a distance" by involving two hosts in the operation. There, we will use a different but equivalent formulation of the rules that isolates this network communication to a single rule.

The `at` modality. The $\Box$ and $\Diamond$ connectives are the only new modalities of Intuitionistic S5. Our first extension to the logic adds a third modality, `at`. The proposition $A \,\texttt{at}\, \mathrm{w}$ means that A is true at the world w; it is a direct internalization of the judgment $A \,\mathsf{true@}\mathrm{w}$ as a proposition [20, 21]. Such internalizations are very common. For instance, implication ($A \supset B$) in propositional logic is the internalization of the hypothetical judgment ($A \,\mathsf{true} \vdash B \,\mathsf{true}$). The rules defining the connective are:

$$\frac{\Gamma \vdash A \,\mathsf{true@}\mathrm{w}'}{\Gamma \vdash A \,\texttt{at}\, \mathrm{w}' \,\mathsf{true@}\mathrm{w}}\ \texttt{at}\,\mathrm{I} \qquad \frac{\Gamma \vdash A \,\texttt{at}\, \mathrm{w}' \,\mathsf{true@}\mathrm{w}}{\Gamma \vdash A \,\mathsf{true@}\mathrm{w}'}\ \texttt{at}\,\mathrm{E}$$

There are several reasons to extend IS5 with the `at` modality. One is that it is a natural and elegant consequence of formalizing IS5 with a world-indexed truth judgment. Because it directly internalizes the machinery of the proof system, it will also give us the ability to make more precise statements. In contrast, $\Box$ and $\Diamond$ are "lossy;" consider the following theorem of IS5:

$$\Box(A \supset B) \supset \Diamond A \supset \Diamond B$$

The reading of this proposition is as follows: Given that A implies B in every world, and given that A is true in some world, B is true in some world. This proposition is true by virtue of the fact that B is true at the same world that A is true (by using the implication). However, the proposition does not state that B is true at the same world; in fact, we unable to state such a thing with the $\Box$ and $\Diamond$ connectives at all. The `at` modality allows us to make such statements. Once we introduce propositions that quantify over worlds, we can say

$$\Box(A \supset B) \supset \forall\omega.(A \,\texttt{at}\, \omega) \supset (B \,\texttt{at}\, \omega)$$

which is stronger than the above. In fact, the $\Box$ and $\Diamond$ modalities are definable in terms of `at` and quantification, where $\Box A$ is $\forall\omega.A \,\texttt{at}\, \omega$ and $\Diamond A$ is $\exists\omega.A \,\texttt{at}\, \omega$. Finally, the precision of the `at` modality will be necessary in the internals of the compiler for the purpose of closure conversion (Section 6.4).

Having completed the background discussion of IS5, we are now prepared to use it as the foundation of the ML5 language, which is presented in the next section. ML5 will modify the logic in several ways; for comparison, all of the standard rules are given in Appendix A.

4 The ML5 Language

I have designed and formalized in Twelf a simple calculus based on the modal logic from the previous section. (The calculus appears as the internal language of the implementation in Section 6.2). This calculus forms the basis of ML5, whose features are toured in this section. However, what is presented here is only that simple calculus dressed up to make it feel like a programming language; it is not a complete description of the language. The specification of ML5 proper is part of the proposed work. Specifically, I do not cover the features that are treated the same as in SML (*e.g.* functions, datatypes, tuples). I also treat issues of concrete syntax loosely.

The language ML5 is designed around a Curry-Howard interpretation of Intuitionistic S5. To do this, we associate proof terms with the rules of the logic. These proof terms become the syntax of the programming language.

Its main typing judgment is

$$\Gamma \vdash M : A@\mathrm{w}$$

which states that the expression M has type A at the world w, in the context Γ. As in the logic, each assumption in Γ is adorned with the world at which it is well-typed. Computationally, these worlds are the hosts in the network.

Because every subexpression of the program is typed at some world, an ML5 program consists of a tree of nested expressions to be evaluated at different hosts. Every variable in the context Γ is bound to a value that the program has access to, and these variables are typed at various different worlds. Therefore the program is able to manipulate both values that make sense to it (ones that are typed at the same world) and values that do not (ones that are typed at different worlds).

The worlds in the typing judgment are the static representations of hosts in the network. At runtime, we will need tokens with which to refer to these worlds. A value of type $\mathrm{w}\,\texttt{addr}$ is such a token, which acts as the address of a host. A host can compute its own address by using the $\texttt{localhost}()$ primitive, which results in an address value of the form $\overline{\mathbf{w}}$:

$$\frac{}{\Gamma \vdash \texttt{localhost}() : \mathrm{w}\,\texttt{addr}\ @\mathrm{w}}\ \text{localhost} \qquad \frac{}{\Gamma \vdash \overline{\mathbf{w}} : \mathbf{w}\,\texttt{addr}\ @\mathrm{w}'}\ \text{address}$$

The way an address is used is by switching the location of evaluation to the world described by the address. This primitive, called get, is ML5's only mechanism for moving control and data between hosts.

$$\frac{\begin{array}{l}\Gamma \vdash \mathrm{w}'\ \textsf{world}\\ \Gamma \vdash M : \mathrm{w}'\,\texttt{addr}@\mathrm{w}\\ \Gamma \vdash N : A@\mathrm{w}'\\ A\ \textsf{mobile}\end{array}}{\Gamma \vdash \texttt{get}[\mathrm{w}'; M]N : A@\mathrm{w}}\ \text{get}$$

The get primitive transfers control from the current world w to the specified world w'. The judgment $\Gamma \vdash \mathrm{w}'\ \textsf{world}$ checks that the world expression w' is well-formed (Figure 2). The expression N, which is well-typed at w', is evaluated there. The resulting value is then returned to w. For an arbitrary value, this motion is unsound; the value may be, or make use of, a resource that is local to w'. The final typing condition for get therefore restricts the primitive to types that are *mobile*. A mobile type is one whose values are always portable between worlds (Figure 1). Primitive types such as integers and strings are mobile. Addresses of worlds are mobile, as are data structures built from other

$$\frac{}{b \text{ mobile}} \qquad \frac{A \text{ mobile} \quad B \text{ mobile}}{A \times B \text{ mobile}}$$

$$\frac{}{\Box A \text{ mobile}} \qquad \frac{}{\Diamond A \text{ mobile}}$$

$$\frac{\begin{array}{c}\alpha \text{ mobile} \\ \vdots \\ A \text{ mobile}\end{array}}{\mu\alpha.A \text{ mobile}} \qquad \frac{A \text{ mobile} \quad B \text{ mobile}}{A + B \text{ mobile}}$$

$$\frac{}{\mathrm{w}\ \texttt{addr} \ \text{ mobile}}$$

Figure 1: The definition of the mobile judgment. The metavariable b ranges over ML base types like `int` and `string`. The rule for recursive types $\mu\alpha.A$ introduces a mobility assumption for the type variable, so mobility is actually a hypothetical judgment. However, no other type variables are ever considered mobile

$$\frac{}{\Gamma, \omega \text{ world}, \Gamma' \vdash \omega \text{ world}} \qquad \frac{}{\Gamma \vdash \mathbf{w} \text{ world}}$$

Figure 2: Definition of the judgment $\Gamma \vdash \mathrm{w}$ world

mobile types. Abstract types, mutable references, and functions are not mobile, among other things. Some functions and abstract types may still be moved, however, by embedding them in the $\Box$ modality, which is mobile.

Local action. The presence of the `get` primitive allows us to restrict all other operations to dynamically involve only one world. In doing so, we will retain all of the expressive power of the logic.

In the logic, the proposition $\Diamond A$ meant that A is true at some world. In the ML5 language, a value of type $\Diamond A$ is an address for some value of type A, which lives at an arbitrary unknown world. The rules for $\Diamond$ in ML5 are as follows:

$$\frac{\Gamma \vdash M : A@\mathrm{w}}{\Gamma \vdash \texttt{here}\ M : \Diamond A@\mathrm{w}} \ \Diamond \mathrm{I}$$

$$\frac{\begin{array}{l}\Gamma \vdash M : \Diamond A@\mathrm{w} \\ \Gamma, \omega \text{ world}, a{:}\omega\ \texttt{addr}@\mathrm{w}, x{:}A@\omega \vdash N : C@\mathrm{w}\end{array}}{\Gamma \vdash \texttt{letd}\ \langle \omega, a, x\rangle = M\ \texttt{in}\ N : C@\mathrm{w}} \ \Diamond \mathrm{E}$$

For both of these primitives, all of the computation happens at the world w. We may only compute the address of the result of evaluating a local expression ($\Diamond$ I). To use something of type $\Diamond A$, we must have it at the current world ($\Diamond$ E). In the body of the `letd`, we have in scope a hypothetical world ω where the value is, an address a for that world, and the value itself x. In comparison, the rules from Section 3 allowed us to evaluate and take the address of remote expressions, as well as eliminate remote addresses. We can derive those rules by composing the ML5 rules with `get`.

In the logic, proposition $\Box A$ meant that A is true in every world. In ML5, a value of type $\Box A$ is a suspended expression of type A that can be evaluated anywhere. The rules for the $\Box$ constructor are also purely local:

$$\frac{\Gamma, \omega \textsf{ world} \vdash M : A@\omega}{\Gamma \vdash \texttt{box}\,\omega.M : \Box A@\mathrm{w}}\ \Box\mathrm{I} \qquad \frac{\Gamma \vdash M : \Box A@\mathrm{w}}{\Gamma \vdash \texttt{unbox}\,M : A@\mathrm{w}}\ \Box\mathrm{E}$$

Although the body of the `box` is typed at the hypothetical world ω ($\Box$ I), this construct does not access any remote worlds at runtime. The code inside the `box` is suspended, so $\texttt{box}\,\omega.M$ is already a value. Unboxing is straightforward ($\Box$ E); the suspended expression is evaluated at this time. As before, the remote version of `unbox` can be derived by composing this rule with `get`.

We now have enough to build some small examples, which we will use to motivate the addition of a new judgment for global reasoning. The following proposition is one of the characteristic axioms of Intuitionistic S5:

$$\Diamond\Box A \supset A$$

As a type it describes a function that when given the address of some suspended portable code of type A we computes a value of type A. In ML5, a function `f` of this type is:

```
fun f (x : ◇□α) : α =
  letd <w, a, y> = x
  in
     unbox(get [w;a] y)
  end
```

The function `f` works by getting the portable code from the world indicated by the address, and then running it locally.

A different characteristic axiom is problematic:

$$\Box A \supset \Box\Box A$$

In ML5, we might like to write this as follows (supposing we are at the world `w`):

```
(* ill-typed *)
fun g (x : □α) : □□α =
  let val a : w addr = localhost()
  in  box w'. get[w; a] x
  end
```

This code does not type-check. The idea is that we save the address of the local machine as `a`. We then create a second box, whose contents is an expression that `get`s the original box from the world `w`. The reason that this does not type-check is that the variable `a` is bound at the world `w`. The `get`, which happens at the world `w'` but tries to use `a`, is ill-typed. Moreover, we cannot `get` the address itself from `w`, since to do so would require an address for `w` at `w'`, which is the very thing we lack! The solution will be to add a new judgment to classify values that are well-typed at any world. Addresses will be such values, which will make the above program well-typed with a small modification. The program also has a second problem: the boxed term that we create has to contact the home world `w` every time it is `unbox`ed. Using the new judgment we will also be able to create a more efficient implementation of this function.

4.1 Validity

To enable global reasoning, we add a new judgment to the language:

$$\Gamma \vdash S \sim A@\mathrm{w}$$

In this judgment, called the validity judgment, S is a *valid expression* with type A at the world w. S will evaluate (at w) to a value that has type A in any world. Assumptions of the form $u{\sim}A$ now appear in Γ as well. Since these assumptions stand for values that are well-typed in any world, they are not annotated with any world at all. A valid hypothesis can be used at any world:

$$\frac{}{\Gamma, u{\sim}A, \Gamma' \vdash u : A@\mathrm{w}}\ \text{vhyp}$$

Valid expressions take on one of the two following forms:

$$\begin{array}{lcll} \text{valid expressions} & S & ::= & \texttt{value}\ \omega.v \\ & & | & \texttt{mobile}\ M \end{array}$$

The typing conditions are as follows:

$$\frac{\Gamma, \omega\ \textsf{world} \vdash v : A@\omega}{\Gamma \vdash \texttt{value}\ \omega.v \sim A@\mathrm{w}}\ \text{value} \qquad \frac{\begin{array}{c} A\ \textsf{mobile} \\ \Gamma \vdash M : A@\mathrm{w} \end{array}}{\Gamma \vdash \texttt{mobile}\ M \sim A@\mathrm{w}}\ \text{mobile}$$

For `value`, the argument is already a value, which must be well-typed in any world. The value is therefore parameterized by a static world ω, and checked for well-typedness in that same world. For `mobile`, the expression must be well-typed, and the type of the expression must be mobile. This ensures that after the expression is evaluated, the result will be well-typed at any world.

The Validity Modality. The valid judgment is internalized to define the circle modality $\bigcirc A$. A value of type $\bigcirc A$ is a suspended expression that, when evaluated, produces a value that is well typed anywhere [34]. It is defined as follows:

$$\frac{\Gamma \vdash S \sim A@\mathrm{w}}{\Gamma \vdash \texttt{cir}\ S : \bigcirc A@\mathrm{w}}\ \bigcirc\text{I} \qquad \frac{\begin{array}{c} \Gamma \vdash M : \bigcirc A@\mathrm{w} \\ \Gamma, u{\sim}A \vdash N : C@\mathrm{w} \end{array}}{\Gamma \vdash \texttt{letv}\ u = M\ \texttt{in}\ N : C@\mathrm{w}}\ \bigcirc\text{E}$$

Because the valid expression is suspended, `cir` S is a value. At runtime, the `letv` construct proceeds by evaluating M to the form `cir value` v or `cir mobile` M, then evaluating M in the latter form, and then binding the resulting value to u for the purpose of evaluating N.

The $\bigcirc$ modality allows us to type the above attempt at implementing $\Box A \supset \Box\Box A$:

```
fun g (x : □α) : □□α =
  letv u ∼ w addr = cir (mobile (localhost()))
  in
     box w'. get[w; u] x
  end
```

In this version, we are able to use the address for w at the world w′ because it is a global hypothesis. A shortcoming remains: the box produced in this version must contact the world w every time it is unboxed. An improved version uses the fact that $\Box\alpha$ is mobile to create a valid hypothesis v from the argument x, which can then be used directly at w′:

```
fun g (x : □α) : □□α =
  letv v ∼ □α = cir (mobile x)
  in
     box w'. v
  end
```

The type $\bigcirc A$ is not mobile, because the suspended expression of type A is typed with respect to the world where the circle is introduced. A portable suspended expression that produces a portable value would have type $\Box\bigcirc A$.

A common idiom is to introduce the circle modality and then immediately eliminate it in order to bind a valid variable. ML5 therefore provides two derived forms:

$$\begin{array}{lcl} \texttt{putv}\, u\, \omega = v\ \texttt{in}\ N & \doteq & \texttt{letv}\ u = \texttt{cir}(\texttt{value}\ \omega.v)\ \texttt{in}\ N \\ \texttt{putm}\, u = M\ \texttt{in}\ N & \doteq & \texttt{letv}\ u = \texttt{cir}(\texttt{mobile}\ M)\ \texttt{in}\ N \end{array}$$

Polymorphism. Like Standard ML, ML5 supports prenex parametric polymorphism over types. It also supports polymorphism over worlds in an analogous way. We create a new syntactic class of polytypes, and two new forms of hypothesis that can appear in Γ:

$$\begin{array}{lll} \text{polytypes} & P & ::= \forall \alpha_1, \ldots, \alpha_n, \omega_1, \ldots, \omega_m . A \\ \text{contexts} & \Gamma & ::= \ldots \mid x{:}P\,@\,\mathrm{w} \mid x{\sim}P \end{array}$$

Only assumptions can have polymorphic type. Every use of a polymorphic assumption must completely instantiate the polymorphic type:

$$\frac{}{\begin{array}{l}\Gamma, x{:}(\forall \alpha_1, \ldots, \alpha_n, \omega_1, \ldots, \omega_m . A)\,@\,\mathrm{w}, \Gamma' \vdash \\ x\langle A_1, \ldots, A_n, \mathrm{w}_1, \ldots, \mathrm{w}_m\rangle : ([\,{}^{A_1}/_{\alpha_1}, \ldots, {}^{A_n}/_{\alpha_n}, {}^{\mathrm{w}_1}/_{\omega_1}, \ldots, {}^{\mathrm{w}_m}/_{\omega_m}]A)\,@\,\mathrm{w}\end{array}}\ \text{polyhyp}$$

Polymorphic variables are introduced by additional polymorphic versions of the `val`, `fun`, and `letv` bindings. For instance, the additional `val` rule is as follows:

$$\frac{\begin{array}{l}\Gamma, \alpha_1 \ \mathsf{type}, \ldots, \alpha_n \ \mathsf{type}, \omega_1 \ \mathsf{world}, \ldots, \omega_m \ \mathsf{world} \vdash v : A @ \mathrm{w} \\ \Gamma, x{:}(\forall \alpha_1, \ldots, \alpha_n, \omega_1, \ldots, \omega_m . A) @ \mathrm{w} \vdash N : C @ \mathrm{w}\end{array}}{\Gamma \vdash \texttt{let} \ \langle \alpha_1, \ldots, \alpha_n ; \omega_1, \ldots, \omega_m \rangle \ \texttt{val} \ x = v \ \texttt{in} \ N : C @ \mathrm{w}} \ \mathrm{val}$$

ML5 has a value restriction, like SML: A binding can only be polymorphic if the right hand side is a value. This allows the type inference process to insert quantifiers without changing the dynamic meaning of the program. This proposal does not include a description of type inference, but I expect it to be a straightforward extension of the standard algorithm.

The `at` Modality. A value of type $A \ \texttt{at} \ \mathrm{w}$ is an encapsulated value, where that value is well-typed at another world.

$$\frac{\Gamma \vdash v : A @ \mathrm{w}'}{\Gamma \vdash \texttt{hold}_{\mathrm{w}'} \ v : A \ \texttt{at} \ \mathrm{w}' @ \mathrm{w}} \ \texttt{at}\,\mathrm{I} \qquad \frac{\begin{array}{c}\Gamma \vdash M : A \ \texttt{at} \ \mathrm{w}' @ \mathrm{w} \\ \Gamma, x{:}A @ \mathrm{w}' \vdash N : C @ \mathrm{w}\end{array}}{\Gamma \vdash \texttt{leta} \ x = M \ \texttt{in} \ N : C @ \mathrm{w}} \ \texttt{at}\,\mathrm{E}$$

The `leta` binding consumes a value of type $A \ \texttt{at} \ \mathrm{w}'$ by putting an assumption $x{:}A @ \mathrm{w}'$ in the context. The value $\texttt{hold}_{\mathrm{w}'} v$ represents the encapsulation of v. Its body must be a value because we are not in general at the correct world to evaluate it; consider the following ill-formed code:

```
(* ill-typed *)
let val x = hold[w'] (ref 0)
in leta y = x
   in
      M
   end
end
```

If this code is to be allowed, then within the expression M, `y` should be bound to an allocated reference at `w'`. There has been no opportunity to create such a reference, however. The static world `w'` might not even have a dynamic address available at this point, which would be necessary to make contact with it.

Dynamically, `hold` is just a type coercion and `leta` behaves just like `let`.

Sums. Sum types, which come from the disjunctive connective in logic, are not entirely straightforward. Like the elimination form for $\Diamond$, the standard logical rule for disjunction allows the remote elimination of $A \vee B$:

$$\frac{\begin{array}{l}\Gamma \vdash A \vee B \ \mathsf{true} @ \mathrm{w}' \\ \Gamma, A \ \mathsf{true} @ \mathrm{w}' \vdash C \ \mathsf{true} @ \mathrm{w} \\ \Gamma, B \ \mathsf{true} @ \mathrm{w}' \vdash C \ \mathsf{true} @ \mathrm{w}\end{array}}{\Gamma \vdash C \ \mathsf{true} @ \mathrm{w}} \ \vee\mathrm{E}$$

In this rule $A \vee B$ may be at a different world (w') than our current one (w). So far, all rules in ML5 other than `get` have acted solely on local data. We would like to maintain this, so that (for one thing) existing single-world ML code still runs without any overhead from the distributed features. The local rules for sum types in ML5 are:

$$\frac{\begin{array}{l}\Gamma \vdash M : A + B @ \mathrm{w} \\ \Gamma, x{:}A @ \mathrm{w} \vdash N_1 : C @ \mathrm{w} \\ \Gamma, x{:}B @ \mathrm{w} \vdash N_2 : C @ \mathrm{w}\end{array}}{\Gamma \vdash \begin{array}{l}\texttt{case}\ M\ \texttt{of} \\ \quad \texttt{inl}\ x \Rightarrow N_1 \\ \ \mid \texttt{inr}\ x \Rightarrow N_2\end{array} : C @ \mathrm{w}} +\mathrm{E}$$

$$\frac{\Gamma \vdash M : A @ \mathrm{w}}{\Gamma \vdash \texttt{inl}\ M : A + B @ \mathrm{w}} +\mathrm{I}_1 \qquad \frac{\Gamma \vdash M : B @ \mathrm{w}}{\Gamma \vdash \texttt{inr}\ M : A + B @ \mathrm{w}} +\mathrm{I}_2$$

For $\Box$ and $\Diamond$, the `get` construct makes the nonlocal logical rules derivable. This does not work as easily for disjunction, because we cannot `get` a value of arbitrary sum type ($A \vee B$ is only mobile if both A and B are mobile.) However, the nonlocal case analysis is still derivable in ML5. Here is one way to do it; the rule

$$\frac{\begin{array}{l}\Gamma \vdash O : \mathrm{w}'\ \texttt{addr}\ @ \mathrm{w} \\ \Gamma \vdash M : A + B @ \mathrm{w}' \\ \Gamma, x{:}A @ \mathrm{w}' \vdash N_1 : C @ \mathrm{w} \\ \Gamma, x{:}B @ \mathrm{w}' \vdash N_2 : C @ \mathrm{w}\end{array}}{\Gamma \vdash \begin{array}{l}\texttt{case}[\mathrm{w}'; O]\ M\ \texttt{of} \\ \quad \texttt{inl}\ x \Rightarrow N_1 \\ \ \mid \texttt{inr}\ x \Rightarrow N_2\end{array} : C @ \mathrm{w}} +\mathrm{E\text{-}nonlocal}$$

can be implemented as

```
let
  val mr = get[w'; O]
              (case M of
                 inl y => inl (hold[w'] y)
               | inr y => inr (hold[w'] y))
in
  case mr of
    inl x' => leta x = x'
              in N1
              end
  | inr x' => leta x = x'
              in N2
              end
end
```

Here, the type of value that we `get` is $A\,\texttt{at}\,\omega' + B\,\texttt{at}\,\omega'$, which is guaranteed to be mobile. Therefore, we retain all of the strength of the logical $\vee$ connective in IS5. The only downside is that this derivation requires us to enlist the `at` modality. The consequence is that the logical completeness relies on the presence of both $\vee$ and `at`, meaning that the two are not truly orthogonal in the language.

Like in Standard ML, labeled sums will actually be supported (along with recursive types) as part of the datatype mechanism.

Properties. The language presented here should have two main properties. First is type safety: Every well-typed program can take a step to another well-typed program, or is already a value. I have proved type safety in Twelf for a simplified but representative version of the calculus. Second is a connection to the logic: the modifications that we have made (particularly the use of `get` and locally-acting rules) should not have reduced the expressive power of the language. We prove for two related lambda calculi [31, 30] that this alternate formulation proves exactly the same theorems as the standard presentation of the logic. Both of these proofs have also been formalized in Twelf. Part of the thesis research will be extending this result to the pure subset of ML5 (with the $\bigcirc$ and `at` modalities) as well.

5 Code Examples

The following small examples illustrate how the constructs of ML5 can be used to build distributed applications.

Mobile code proxy. Many distributed programming languages provide a way of forming "proxies" for code that cannot itself move. These proxies can then be passed around, and used as stand-ins for the functions they mimic. In ML5, if the argument and return types of a function are mobile, then one can create a proxy for it. Here we'll assume we have a variable $f{:}A \to B\,@\,\mathbf{w}_0$ and will bind a valid variable $u{\sim}A \to B$ which is the mobile proxy for that function. This assumes that A and B are mobile.

```
putm w0a = localhost()
in putv u w' =
     (fn a =>
      putm v = a
      in
        get [w0; w0a] (f v)
      end)
   in
      ...
   end
end
```

We begin by computing the address of $\mathbf{w}_0$ and making it valid, so that we can contact $\mathbf{w}_0$ from any world. We then bind u to a function value, which is the proxy. The proxy works by taking the argument (known to be mobile by assumption) and making it valid as well. The proxy then performs a `get` to apply the original function at $\mathbf{w}_0$ and return the result.

Cords. Assuming some simple facilities for concurrency and the discovery of idle hosts, we can also emulate the features of Grid/ML in a statically-typed way. The Grid/ML client will be world h, which is the world that the program begins running on in ML5. Let's suppose that the following are bound in the context:

$$\begin{gathered}\texttt{future}::\mathsf{Type} \to \mathsf{Type},\\ \texttt{future} \sim ((\texttt{unit} \to \alpha) \to \alpha\,\texttt{future}),\\ \texttt{now} \sim (\alpha\,\texttt{future} \to \alpha),\\ \texttt{find\_idle} \sim (\texttt{unit} \to \Diamond\texttt{unit})\end{gathered}$$

We assume Multilisp-like futures, which are a simple mechanism for fork-join parallelism. An α `future` is a computation, running concurrently, that produces a value of type α when synchronized on with `now`. Futures must allow the execution of arbitrary ML5 code, but needn't have any special features for distribution. We also expect a `find_idle` routine to give the address of an idle host, which is represented here as $\Diamond$`unit`. All of these primitives are valid hypotheses, so they can be used in any world.

With this, we can implement a "mobile `spawn`" function like the one that is provided by Grid/ML.

```
(* spawn ~ ∀α,ω.(□(α at ω) -> (α at ω) future) *)
putv ⟨α, ω⟩ spawn w2 =
       (fn (code : □(α at ω)) =>
        future (fn () =>
                letd <w3, w3a, ()> = find_idle()
                in putm c = code
                   in get[w3; w3a] (unbox c)
                   end
                end))
in
   ...
end
```

This implementation of `spawn` takes as an argument a piece of suspended code suitable to run at any world. That suspended code must produce a result that is well-typed at the world ω. The function `spawn` is polymorphic in ω, so this is the caller's choice. It then produces a future that finds an idle host and runs the code there.

The following code shows how `spawn` could be used from the home world h:

```
val f = spawn⟨h, int⟩
          (box _,_.
            let fun nfactors (n : int) : int = ...
            in  putm res = nfactors n
                in hold[h] res
                end
            end)

(* ... some computations in parallel ... *)
val answer : int = (leta a = now f in a end)
```

This program invokes spawn with some "cord" code. That cord computes the number of factors of some specific n, and coerces the resulting value (an integer, which is mobile) to the world `h`. There is no requirement that the result of the computation be mobile; if we like, it can be specifically tailored to the world that spawns it. Once we've spawned the cord `f`, we can later wait on its answer with `now`. Since it has type `int at h` and we are running at `h`, we can use `leta` to unencapsulate the result into a regular integer.

Although this requires more text than using the built-in primitives of Grid/ML, ML5 has the advantage of superior static checking. The type system ensures that the code given to `spawn` can run anywhere, and ensures that the result is well-formed at the world `h`. Unlike Grid/ML, we can also be more specific about the capabilities of the hosts on which our mobile code will run. Here we have represented the addresses of hosts as $\Diamond$`unit`. If we wanted to express the idea that machines on the Grid have their own distinct set of capabilities, we need only amend the implementation of `find_idle` to return $\Diamond$ `caps` for some appropriate description of its local capabilities.

Having toured the language's features and looked at some simple examples, we now switch gears to a discussion of its implementation.

6 The ML5 Compiler

This section describes the planned implementation of a compiler for ML5. The early phases of the compiler have been studied in detail. The later phases have not; part of the thesis work will be to figure out the best course of action for these and take it.

6.1 Design Issues

Before walking through the phases of the compiler, it is important to recall some of the issues that will guide the design.

Data Representation. The chief thing that makes the implementation of ML5 different from a standard type-directed compiler for ML is that programs are able to manipulate data that are typed at different worlds. Because of our local-only elimination rules, the only way that these data can be used is in preparation of code or data also intended to be evaluated at other worlds. Therefore, the implementation has considerable flexibility in the way these abstract remote values are represented—the choices can be guided almost entirely by the marshalling strategy.

Marshalling. Because ML5 programs run on a network, we require the ability to marshal code and data into raw bytes to be sent over that network. The compiler will support the runtime in this task by providing marshalling and unmarshalling functions for each abstract type. (Recall that any ML5 value can be marshalled, even if the type is not mobile.)

Certification. In order to use ML5 in our trustless Grid computing project, the compiler must produce certificates that demonstrate the safety of the mobile code. I will do this by building a type-directed compiler that preserves and translates types throughout the phases of compilation. The compiler will output typed assembly code for the TALT system [13].

The phases of compilation are as follows. First, the external language (ML5) is elaborated into a simpler internal language (IL) (Section 6.2). Second, we convert this IL to continuation passing style (CPS) (Section 6.3). Next, we perform closure conversion to implement lexically scoped closures with closed functions (Section 6.4). Each of these phases is standard [28] apart from the new features of ML5, so the description concentrates on those.

6.2 Elaboration

The purpose of elaboration is to replace features of convenience in ML5 with the simpler type theoretic constructs of the IL. For the purpose of this proposal, those features are the $\Box$ and $\Diamond$ modalities and world polymorphism.

World Quantification. In the IL, type inference is no longer an issue. We can therefore abandon ML5's restriction to prenex polymorphism, which was expressed in terms of a syntactic class of "polytypes." Instead we simply have the type constructor $\forall\omega.A$:

$$\frac{\Gamma, \omega\ \mathsf{world} \vdash v : A@\mathrm{w}}{\Gamma \vdash \Lambda\omega.v : \forall\omega.A@\mathrm{w}}\ \forall\mathrm{I} \qquad \frac{\Gamma \vdash M : \forall\omega.A@\mathrm{w}}{\Gamma \vdash M\langle\mathrm{w}'\rangle : [\mathrm{w}'/\omega]A@\mathrm{w}}\ \forall\mathrm{E}$$

The IL supports a "type-erasure semantics" [28] where types are purely static entities, and the term constructors that manipulate them can be erased without changing the dynamic meaning of the program. To achieve this, the constructor Λ is restricted to values. This is all we need for the elaboration of ML5, where polymorphism is also restricted to values.

The IL also has a first-class existential quantifier, which will be used for closure conversion and the elaboration of $\diamond$:

$$\frac{\Gamma \vdash \mathrm{w}' \;\mathsf{world} \qquad \Gamma \vdash v : [\mathrm{w}'/\omega]A@\mathrm{w}}{\Gamma \vdash \mathtt{pack}[\mathrm{w}']v \;\mathtt{as}\; \exists\omega.A : \exists\omega.A@\mathrm{w}}\;\exists\,\mathrm{I} \qquad \frac{\Gamma \vdash M : \exists\omega.A@\mathrm{w} \qquad \Gamma, \omega\;\mathsf{world}, x{:}A \vdash N : C@\mathrm{w}}{\Gamma \vdash \mathtt{unpack}\; M \;\mathtt{as}\; \omega, x \;\mathtt{in}\; N : C@\mathrm{w}}\;\exists\,\mathrm{E}$$

The Elaboration Relation. Elaboration is given as a function on ML5 typing derivations. As a shorthand, we write this as a function on expressions, $[\![\cdot]\!]_{\mathsf{E}}^{A@\mathrm{w}}$, parameterized by the current world w and the type A of the expression being translated.[2] We also give a function that translates ML5 types $[\![\cdot]\!]_{\mathsf{E}}$. For the syntactic classes of values and valid expressions, we have have two additional translations $[\![\cdot]\!]_{\mathsf{VE}}^{A@\mathrm{w}}$ and $[\![\cdot]\!]_{\mathsf{SE}}^{A@\mathrm{w}}$. The cases for eliminating the $\Box$ and $\diamond$ modalities are:

$$\begin{array}{lcl}
[\![\Box A]\!]_{\mathsf{E}} & = & \forall\omega.((\mathtt{unit} \rightarrow [\![A]\!]_{\mathsf{E}})\;\mathtt{at}\;\omega) \\
[\![\diamond A]\!]_{\mathsf{E}} & = & \exists\omega.([\![A]\!]_{\mathsf{E}}\;\mathtt{at}\;\omega \times \omega\;\mathtt{addr}) \\
[\![\mathtt{box}\;\omega'.M]\!]_{\mathsf{VE}}^{\Box A@\mathrm{w}} & = & \Lambda\omega'.\mathtt{hold}_{\omega'}\lambda().[\![M]\!]_{\mathsf{E}}^{A@\omega'} \\
[\![\mathtt{unbox}\;M]\!]_{\mathsf{E}}^{A@\mathrm{w}} & = & \mathtt{leta}\;x = [\![M]\!]_{\mathsf{E}}^{\Box A@\mathrm{w}}\langle\mathrm{w}\rangle\;\mathtt{in}\;x()\;\mathtt{end} \\
[\![\mathtt{here}\;M]\!]_{\mathsf{E}}^{\diamond A@\mathrm{w}} & = & \mathtt{let}\;x = [\![M]\!]_{\mathsf{E}}^{A@\mathrm{w}}\;\mathtt{in} \\
 & & \mathtt{let}\;a = \mathtt{localhost}()\;\mathtt{in} \\
 & & \mathtt{pack}[\mathrm{w}](x,a)\;\mathtt{as}\;\exists\omega.([\![A]\!]_{\mathsf{E}}\;\mathtt{at}\;\omega \times \omega\;\mathtt{addr}) \\
[\![\mathtt{letd}\;\langle\omega,a,x\rangle = M\;\mathtt{in}\;N]\!]_{\mathsf{E}}^{C@\mathrm{w}} & = & \mathtt{unpack}\;[\![M]\!]_{\mathsf{E}}^{\diamond\_@\mathrm{w}}\;\mathtt{as}\;\omega, y\;\mathtt{in} \\
 & & \mathtt{let}\;x = \mathtt{fst}\;y\;\mathtt{in} \\
 & & \mathtt{let}\;a = \mathtt{snd}\;y\;\mathtt{in}\;[\![N]\!]_{\mathsf{E}}^{C@\mathrm{w}}
\end{array}$$

World polymorphism in ML5 is translated into uses of the $\forall\omega$ quantifier in the IL. Because polymorphic types are a different syntactic class in ML5, this requires another translation $[\![\cdot]\!]_{\mathsf{E}}^{\forall}$ for polytypes.

$$\begin{array}{lcl}
[\![\forall\alpha_1,\ldots,\alpha_n,\omega_1,\ldots,\omega_m.A]\!]_{\mathsf{E}}^{\forall} & = & \forall\alpha_1.\ldots,\forall\alpha_n.\forall\omega_1,\ldots,\forall\omega_m.[\![A]\!]_{\mathsf{E}} \\
[\![x\langle A_1,\ldots,A_n,\mathrm{w}_1,\ldots,\mathrm{w}_m\rangle]\!]_{\mathsf{E}}^{A@\mathrm{w}} & = & x\langle A_1\rangle\ldots\langle A_n\rangle\langle\mathrm{w}_1\rangle\ldots\langle\mathrm{w}_m\rangle
\end{array}$$

$$\begin{array}{l}
[\![\mathtt{let}\;\mathtt{val}\;\langle\alpha_1,\ldots,\alpha_n,\omega_1,\ldots,\omega_n\rangle x = v\;\mathtt{in}\;N]\!]_{\mathsf{E}}^{C@\mathrm{w}} \\
\qquad = \;\mathtt{let}\;\mathtt{val}\;x = \Lambda\alpha_1.\ldots\Lambda\alpha_n.\Lambda\omega_1.\ldots\Lambda\omega_m.[\![v]\!]_{\mathsf{E}}^{\_@\mathrm{w}} \\
\qquad\quad\;\; \mathtt{in}\;[\![N]\!]_{\mathsf{E}}^{C@\mathrm{w}}
\end{array}$$

Type Polymorphism. One final phase is needed to eliminate the need for types at run-time. The program must be able to marshal values of any type, including the abstract type variables bound by Λ. I will achieve this by using dictionary-passing to make available a marshalling and unmarshalling function for each type variable. This is a standard implementation technique for type classes [48]. Section 6.5 explains marshalling in more detail.

[2]In some recursive invocations of the function, we must produce with the type of a subexpression, where that type is not evident from the expression itself. This is why elaboration works on typing derivations (where all of the types of subterms are present) rather than directly on the expressions. In those recursive calls we write _ for the missing type.

Properties. Elaboration should be type-preserving in the following sense, where $[\![\Gamma]\!]_{\mathsf{E}}$ is the natural extension of $[\![A]\!]_{\mathsf{E}}$ to contexts:

Property 1 (Type soundness of elaboration)

(a) *If* $\Gamma \vdash M : A@\mathrm{w}$ *(in the EL)*
then $[\![\Gamma]\!]_{\mathsf{E}} \vdash [\![M]\!]_{\mathsf{E}}^{A@\mathrm{w}} : [\![A]\!]_{\mathsf{E}}@\mathrm{w}$ *(in the IL).*

(b) *If* $\Gamma \vdash v : A@\mathrm{w}$ *(in the EL)*
then $[\![\Gamma]\!]_{\mathsf{E}} \vdash [\![v]\!]_{\mathsf{VE}}^{A@\mathrm{w}} : [\![A]\!]_{\mathsf{E}}@\mathrm{w}$ *(in the IL).*

(c) *If* $\Gamma \vdash S \sim A@\mathrm{w}$ *(in the EL)*
then $[\![\Gamma]\!]_{\mathsf{E}} \vdash [\![S]\!]_{\mathsf{SE}}^{A@\mathrm{w}} : [\![A]\!]_{\mathsf{E}}@\mathrm{w}$ *(in the IL).*

This property has been formalized and proved in Twelf for a simplified external language without polymorphism [25]. □

6.3 Continuation Passing Style

After elaborating into the IL, the goal of the compiler is to repeatedly simplify the code by removing the use of constructs that have no direct analogue in machine code. The first such translation serializes the evaluation of expressions and makes control flow explicit by putting the program in continuation passing style [4].

Other than the translation of valid expressions and `get`, CPS conversion is standard. Figure 3 contains a representative sample of the CPS language (the dynamic semantics for this fragment are given in Appendix C). A value of type A `cont` is a continuation expecting a value of type A. Its canonical form is $\lambda x.c$. The type ∂A is a new modality that is used to compile the $\bigcirc A$ modality. A value of type ∂A is an encapsulated value that is well-typed anywhere. It is an internalization of the $u{\sim}A$ hypothesis just as the `at` modality is an internalization of the $x{:}A@\mathrm{w}$ hypothesis. The canonical form of type ∂A is `cval` $\omega.v$. The value $\overline{\mathbf{w}}$ represents an address for the world **w**. Such values are introduced by the `localhost` operation.

We use the judgment

$$\Gamma \vdash c@\mathrm{w}$$

to type-check CPS expressions, because they do not return (and so they cannot be distinguished by return type). We use

$$\Gamma \vdash v : A@\mathrm{w}$$

$$
\begin{array}{rrcl}
\text{exps} & c & ::= & \texttt{leta}\, x = v\ \texttt{in}\ c \\
 & & \mid & \texttt{letg}\, u = v\ \texttt{in}\ c \\
 & & \mid & \texttt{lift}\, u = v\ \texttt{in}\ c \\
 & & \mid & \texttt{let}\ x = \texttt{fst}\ v\ \texttt{in}\ c \\
 & & \mid & \texttt{let}\ x = \texttt{snd}\ v\ \texttt{in}\ c \\
 & & \mid & \texttt{let}\ x = \texttt{mkpair}\ v_1, v_2\ \texttt{in}\ c \\
 & & \mid & \texttt{let}\ x = \texttt{localhost}()\ \texttt{in}\ c \\
 & & \mid & \ldots \\
 & & \mid & \texttt{go}[\mathrm{w}; v_a]\ c \\
 & & \mid & \texttt{call}\ v_f(v_x) \\
 & & \mid & \texttt{halt} \\
\text{values} & v & ::= & \langle v_1, v_2 \rangle \\
 & & \mid & \texttt{hold}_{\mathrm{w}} v \\
 & & \mid & \lambda x.c \\
 & & \mid & \Lambda\omega.v \\
 & & \mid & \Lambda\alpha.v \\
 & & \mid & \texttt{cval}\ \omega.v \\
 & & \mid & \overline{\mathbf{w}} \\
 & & \mid & u \\
 & & \mid & x \\
 & & \mid & \ldots \\
\text{types} & A, B & ::= & A\, \texttt{cont} \mid A \times B \mid A\, \texttt{at}\, \mathrm{w} \\
 & & \mid & \Game A \mid \forall\omega.A \mid \exists\omega.A \\
 & & \mid & \mathrm{w}\ \texttt{addr} \mid \ldots
\end{array}
$$

Figure 3: A representative sample of the CPS language

$$
\begin{array}{rcl}
[\![A \to B]\!]_{\mathsf{C}} & = & ([\![A]\!]_{\mathsf{C}} \times ([\![B]\!]_{\mathsf{C}}\, \texttt{cont}))\, \texttt{cont} \\
[\![\bigcirc A]\!]_{\mathsf{C}} & = & (\ni [\![A]\!]_{\mathsf{C}})\, \texttt{cont}\, \texttt{cont}
\end{array}
$$

Figure 4: Selected cases for the CPS type translation

$$\frac{}{b\ \textsf{cmobile}} \qquad \frac{A\ \textsf{cmobile} \quad B\ \textsf{cmobile}}{A \times B\ \textsf{cmobile}}$$

$$\frac{\begin{array}{c}\alpha\ \textsf{cmobile} \\ \vdots \\ A\ \textsf{cmobile}\end{array}}{\mu\alpha.A\ \textsf{cmobile}} \qquad \frac{A\ \textsf{cmobile} \quad B\ \textsf{cmobile}}{A + B\ \textsf{cmobile}}$$

$$\frac{A\ \textsf{cmobile}}{\forall\alpha.A\ \textsf{cmobile}} \qquad \frac{A\ \textsf{cmobile}}{\forall\omega.A\ \textsf{cmobile}}$$

$$\frac{A\ \textsf{cmobile}}{\exists\alpha.A\ \textsf{cmobile}} \qquad \frac{A\ \textsf{cmobile}}{\exists\omega.A\ \textsf{cmobile}}$$

$$\frac{}{\mathrm{w}\ \texttt{addr}\ \textsf{cmobile}}$$

Figure 5: The definition of the cmobile judgment

to type-check values. The interesting typing rules are as follows:

$$\frac{\Gamma \vdash v_a : \mathrm{w}'\ \texttt{addr}@\mathrm{w} \quad \Gamma \vdash c@\mathrm{w}'}{\Gamma \vdash \textsf{go}[\mathrm{w}'; v_a]\ c@\mathrm{w}}\ \text{go}$$

$$\frac{\Gamma \vdash v_f : A\, \texttt{cont}\, @\mathrm{w} \quad \Gamma \vdash v_x : A@\mathrm{w}}{\Gamma \vdash \texttt{call}\ v_f(v_x)@\mathrm{w}}\ \texttt{cont}\ \text{E}$$

$$\frac{\begin{array}{l}A\ \textsf{cmobile} \\ \Gamma \vdash v : A@\mathrm{w} \\ \Gamma, u{\sim}A \vdash c@\mathrm{w}\end{array}}{\Gamma \vdash \texttt{lift}\ u = v\ \texttt{in}\ c@\mathrm{w}}\ \texttt{lift}$$

$$\frac{\Gamma, \omega\ \textsf{world} \vdash v : A@\omega}{\Gamma \vdash \texttt{cval}\ \omega.v : \ni A@\mathrm{w}}\ \ni\text{I}$$

$$\frac{\Gamma \vdash v : \ni A@\mathrm{w} \quad \Gamma, u{\sim}A \vdash c@\mathrm{w}}{\Gamma \vdash \texttt{letg}\ u = v\ \texttt{in}\ c@\mathrm{w}}\ \ni\text{E}$$

In the CPS language, go replaces ML5's `get`. The go construct invokes a continuation at a remote world. In contrast to the IL's `get`, go is not restricted to mobile types, because it is only a control-flow construct. The mobility restriction shows up in `lift`, which is also used in the translation of `get` below.

The `call` primitive is used to pass a value to a continuation. It simply requires that the argument is the same type that the continuation expects, and that both are typed at the same world. The `lift` construct binds a valid variable to a value whose type is cmobile. The cmobile judgment is analogous to the mobile judgment from ML5 and its definition appears in Figure 5. The `cval` construct, which is the introduction form for the $\circlearrowleft$ modality, requires a value. The typing rule checks that the value is well-typed in any world by checking it in a fresh hypothetical world. The elimination form, `letg`, simply binds the valid variable in the continuation.

There are four translations to convert from the IL to CPS, which translate IL types, expressions, values, and valid expressions. The translation of IL types to CPS types is given by a function $[\![\cdot]\!]_{\mathsf{c}}$. The interesting cases are given in Figure 4. An important lemma states that this translation preserves our notion of mobility:

Lemma 1 (Preservation of Mobility)
If A mobile *then* $[\![A]\!]_{\mathsf{c}}$ cmobile*.*

Proof is straightforward by induction on the derivation of A mobile. □

The translation of IL terms to CPS terms is given in terms of a higher-order function convert. It takes as arguments the current world, an IL expression, and a meta-level continuation to which the result of converting the IL expression is passed. For example, the translation for tuple construction is as follows:

```
convert w $\langle M_1, M_2 \rangle$ $\mathcal{K}$ =
  convert w $M_1$ $\mathcal{K}_1$
    where $\mathcal{K}_1(v_1)$ =
      convert w $M_2$ $\mathcal{K}_2$
        where $\mathcal{K}_2(v_2)$ =
          let $x$ = mkpair $v_1, v_2$
          in $\mathcal{K}(x)$
```

The translation of IL values is given by a second function, $[\![\cdot]\!]^{\mathrm{w}}_{\mathsf{cv}}$, which produces a CPS value. This is standard; we give only a few cases:

$$\begin{array}{rcl} [\![n]\!]^{\mathrm{w}}_{\mathsf{cv}} & = & n \\ [\![\lambda x.M]\!]^{\mathrm{w}}_{\mathsf{cv}} & = & \lambda y.\mathtt{let}\ x = \mathtt{fst}\ y\ \mathtt{in} \\ & & \quad \mathtt{let}\ k = \mathtt{snd}\ y\ \mathtt{in} \\ & & \quad \mathsf{convert}\ \mathrm{w}\ M\ \mathcal{K} \\ & & \quad\quad \text{where}\ \mathcal{K}\ v = \mathtt{call}\ k(v) \\ & \vdots & \\ [\![\mathtt{cir}\ S]\!]^{\mathrm{w}}_{\mathsf{cv}} & = & [\![S]\!]^{\mathrm{w}}_{\mathsf{cs}} \end{array}$$

The case for `cir` makes use of the third translation function, $[\![\cdot]\!]^{\mathrm{w}}_{\mathsf{cs}}$. It takes a valid expression of type A and the current world, and produces a CPS value of type $(\eth[\![A]\!]_{\mathsf{c}})$ `cont cont`. This function is defined as follows:

$$\begin{array}{rcl}
[\![\texttt{valid}\,\omega.v]\!]^{\mathrm{w}}_{\mathsf{cs}} & = & \lambda k.\texttt{call}\;k(\texttt{cval}\;\omega.v) \\
[\![\texttt{mobile}\;M]\!]^{\mathrm{w}}_{\mathsf{cs}} & = & \lambda k.\mathsf{convert}\;\mathrm{w}\;M\;\mathcal{K} \\
 & & \text{where } \mathcal{K}(v) = \\
 & & \quad \texttt{lift}\;u = v\;\texttt{in} \\
 & & \quad \texttt{call}\;k(\texttt{cval}\;\omega.u)
\end{array}$$

We can now give the interesting cases of convert. The elimination form for $\bigcirc$ is translated into a use of `letg`:

```
convert w (letv $u = M$ in $N$) $\mathcal{K}$ =
  convert w $M$ $\mathcal{K}_1$
    where $\mathcal{K}_1(v)$ = call $v(\lambda x.$ letg $u = x$ in
                                       convert w $N$ $\mathcal{K})$
```

The result of converting M (of type $\bigcirc A$) is a value of type $(\eth[\![A]\!]_{\mathsf{c}})$ `cont cont`. We call this continuation with a continuation that unpacks its argument into a validity variable and proceeds with the translated body of the `letv`.

The other interesting case is the translation of `get`:

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

The translation consists of two uses of `go`; one to transfer control to the remote world, and one to come back. We start by evaluating M_a to the value $v_{a'}$, the address for the remote world. We then compute the local address a, which will be used to get back. Because we need to use this address from the remote world, we create a valid hypothesis u_a from it with `lift` (addresses are mobile). We then transfer control to the world w' with `go`. While in the world w', the translation of N is evaluated. Since we want to use the result v_r back at the original world w, it is made into a valid hypothesis using `lift` again. The typing condition for the IL `get` construct requires A to be mobile, so by Lemma 1 the type of this value is cmobile as required. The code then returns to the world w to continue with the final result.

Properties. The CPS translation should satisfy the following properties, where $[\![\Gamma]\!]_{\mathsf{c}}$ is the natural extension of $[\![A]\!]_{\mathsf{c}}$ to contexts:

Property 2 (Type soundness of CPS translation)

(a) *If* $\Gamma \vdash v : A@\mathrm{w}$ *(in the IL)*
then $[\![\Gamma]\!]_{\mathsf{c}} \vdash [\![v]\!]^{\mathrm{w}}_{\mathsf{cv}} : [\![A]\!]_{\mathsf{c}}@\mathrm{w}$ *(in CPS).*

(b) *If* $\Gamma \vdash S : A@\mathrm{w}$ *(in the IL)*
then $[\![\Gamma]\!]_{\mathsf{c}} \vdash [\![S]\!]^{\mathrm{w}}_{\mathsf{cs}} : (\ni[\![A]\!]_{\mathsf{c}})\,\texttt{cont cont}\,@\mathrm{w}$ *(in CPS).*

(c) *If* $\Gamma \vdash M : A@\mathrm{w}$ *(in the IL)*
and for all v *such that* $[\![\Gamma]\!]_{\mathsf{c}} \vdash v : [\![A]\!]_{\mathsf{c}}@\mathrm{w}$, $[\![\Gamma]\!]_{\mathsf{c}} \vdash \mathcal{K}(v)@\mathrm{w}$ *(in CPS)*
then $[\![\Gamma]\!]_{\mathsf{c}} \vdash (\mathsf{convert}\ \mathrm{w}\ M\ \mathcal{K})@\mathrm{w}$ *(in CPS).*

This property has been proven and formalized in Twelf for a similar language without the validity judgment, ○, and ∋ modalities [25]. □

6.4 Closure Conversion

The next important step in compilation is closure conversion. Although low-level machine languages have the ability to represent higher-order code (by passing addresses of code), there is no direct support for nesting these code blocks within one another with lexical scope. Closure conversion implements lexical scope by constructing an explicit *environment* for each λ, so that each function is actually closed.

Once every function is closed, we can hoist the definitions to the top level, and refer to them by label rather than variable. In our distributed setting, the idea is that each label globally identifies a code block, so that we need only to pass around labels and environments in order to implement mobile code. To do this, every host must have the code for every label that it might be asked to execute; one conservative approximation simply gives every host access to all code in the program.

Closure conversion in this setting is conceptually standard, but is complicated by the presence of valid hypotheses and hypotheses at worlds other than the current world. For instance, the following (IL) function at world $\mathbf{w}_0$ has a free variable at $\mathbf{w}_1$:

$$a{:}\mathbf{w}_1\ \texttt{addr@}\mathbf{w}_0, x{:}\texttt{int}\ \texttt{@}\mathbf{w}_1 \vdash \lambda y.y + \texttt{get}[\mathbf{w}_1; a]x : \texttt{int} \rightarrow \texttt{int@}\mathbf{w}_0$$

To generate an environment for this function, we need to reify the remote hypothesis as a value. Because it does not make sense here, we do this by using the `at` modality.

The destination language of closure conversion is the same as for CPS, except for the typing rule for the `go` construct. For the sake of economy, we reuse the CPS language and add a new construct `go_cc` which we use in place of the old.

The closure conversion algorithm is given by three functions. One converts CPS types to CPS types. The only types that are affected by the translation are function (continuation) types. The type of a converted continuation will be an existential:

$$[\![A\ \texttt{cont}]\!]_{\mathsf{cc}} = \exists\alpha_{\text{env}}.\alpha_{\text{env}} \times (([\![A]\!]_{\mathsf{cc}} \times \alpha_{\text{env}})\ \texttt{cont})$$

The other two functions translate CPS expressions and values, respectively. For the translation of values, only lambdas are affected:[3]

$$\begin{array}{rcl}
[\![\lambda x.c]\!]_{\mathsf{CC}} & = & \mathtt{pack}\ \langle B_{\text{env}}, \langle v_{\text{env}}, \lambda p.c'\rangle\rangle \\
 & & \mathtt{as}\ \exists\alpha_{\text{env}}.\alpha_{\text{env}} \times (([\![A]\!]_{\mathsf{CC}} \times \alpha_{\text{env}})\ \mathtt{cont}) \\
\textit{where}\ldots & & \\
c' & = & \mathtt{let}\ x = \pi_2\ p\ \mathtt{in} \\
 & & \mathtt{let}\ e = \pi_1\ p\ \mathtt{in} \\
 & & \mathtt{leta}\ \mathsf{FV}(c)_1 = \pi_1 e\ \mathtt{in} \\
 & & \qquad\vdots \\
 & & \mathtt{leta}\ \mathsf{FV}(c)_n = \pi_n e\ \mathtt{in} \\
 & & \mathtt{letg}\ \mathsf{FSV}(c)_1 = \pi_{n+1} e\ \mathtt{in} \\
 & & \qquad\vdots \\
 & & \mathtt{letg}\ \mathsf{FSV}(c)_m = \pi_{n+m} e\ \mathtt{in} \\
 & & [\![c]\!]_{\mathsf{CC}} \\
v_{\text{env}} & = & \langle \mathtt{hold}_{\mathsf{FVW}(c)_1}(\mathsf{FV}(c)_1), \ldots, \mathtt{hold}_{\mathsf{FVW}(c)_n}(\mathsf{FV}(c)_n), \\
 & & \ \mathtt{cval}\ \omega'.(\mathsf{FSV}(c)_1), \ldots, \mathtt{cval}\ \omega'.(\mathsf{FSV}(c)_m)\rangle \\
B_{\text{env}} & = & [\![\mathsf{FVT}(c)_1]\!]_{\mathsf{CC}}\ \mathtt{at}\ \mathsf{FVW}(c)_1 \times \ldots \times [\![\mathsf{FVT}(c)_n]\!]_{\mathsf{CC}}\ \mathtt{at}\ \mathsf{FVW}(c)_n \times \\
 & & \ \eth(\mathsf{FSVT}(c)_1) \times \ldots \times \eth(\mathsf{FSVT}(c)_m) \\
n & = & \text{Number of free regular variables in } c. \\
\mathsf{FV}(c)_i & = & \text{The } i^{\text{th}} \text{ free regular variable of } c. \\
\mathsf{FVT}(c)_i & = & \text{The type of the } i^{\text{th}} \text{ free regular variable of } c. \\
\mathsf{FVW}(c)_i & = & \text{The world of the } i^{\text{th}} \text{ free regular variable of } c. \\
m & = & \text{Number of free valid variables in } c. \\
\mathsf{FSV}(c)_i & = & \text{The } i^{\text{th}} \text{ free valid variable of } c. \\
\mathsf{FSVT}(c)_i & = & \text{The type of the } i^{\text{th}} \text{ free valid variable of } c.
\end{array}$$

Note that the translation of $\lambda x.c$, a value, is also a value. Because we have lasting restrictions on the bodies of *e.g.* `hold`, it is important to preserve the syntactic classes of values and expressions in these translations.

The translation of expressions only affects `call` and `go`:

$$\begin{array}{rcl}
[\![\mathtt{call}\ v_f(v_x)]\!]_{\mathsf{CC}} & = & \mathtt{unpack}\ [\![v_f]\!]_{\mathsf{CC}}\ \mathtt{as}\ \langle\alpha_{\text{env}}, p\rangle\ \mathtt{in} \\
 & & \mathtt{let}\ f = \mathtt{fst}\ p\ \mathtt{in} \\
 & & \mathtt{let}\ e = \mathtt{snd}\ p\ \mathtt{in} \\
 & & \mathtt{let}\ a = \mathtt{mkpair}[\![v_x]\!]_{\mathsf{CC}}, e\ \mathtt{in} \\
 & & \mathtt{call}\ f(a) \\
[\![\mathtt{go}[\mathrm{w}; v_a]c]\!]_{\mathsf{CC}} & = & \mathtt{go\_cc}[\mathrm{w}, [\![v_a]\!]_{\mathsf{CC}}][\![\lambda\_.c]\!]_{\mathsf{CC}}
\end{array}$$

The translation of `call` is a standard closure call. We unpack the closure, build the argument to the function (which consists of the original argument along with the function's environment) and call it. The translation of `go` is necessary

[3]Here we assume the language has n-ary products with projections π_i.

in order to explicitly represent the continuation as a closure, which is what is transmitted over the network in the implementation. The typing rule for go_cc is:

$$\frac{\Gamma \vdash v_a : \mathrm{w}' \;\texttt{addr}@\mathrm{w} \quad \Gamma \vdash v_k : \exists \alpha_{\mathrm{env}}.\alpha_{\mathrm{env}} \times ((\langle\rangle \times \alpha_{\mathrm{env}})\;\texttt{cont})@\mathrm{w}'}{\Gamma \vdash \texttt{go\_cc}[\mathrm{w}'; v_a]\; v_k @\mathrm{w}}\;\text{go\_cc}$$

Properties. Closure conversion should admit the following properties:

Property 3 (Soundness of closure conversion)

(a) *If* $\Gamma \vdash v : A@\mathrm{w}$ *(in CPS)*
then $[\![\Gamma]\!]_{\mathsf{CC}} \vdash [\![v]\!]_{\mathsf{CC}} : [\![A]\!]_{\mathsf{CC}}@\mathrm{w}$ *(in CPS).*

(b) *If* $\Gamma \vdash c@\mathrm{w}$ *(in CPS)*
then $[\![\Gamma]\!]_{\mathsf{CC}} \vdash [\![c]\!]_{\mathsf{CC}}@\mathrm{w}$ *(in CPS).*

(c) *If* $\Gamma \vdash v : A@\mathrm{w}$ *(in CPS)*
then for every subterm v' *of the form* $\lambda x.c$ *within* $[\![v']\!]_{\mathsf{CC}}$,
$\mathsf{FV}(v') = \mathsf{FSV}(v') = \emptyset$.

(d) *If* $\Gamma \vdash c@\mathrm{w}$ *(in CPS)*
then for every subterm v *of the form* $\lambda x.c'$ *within* $[\![c]\!]_{\mathsf{CC}}$,
$\mathsf{FV}(v) = \mathsf{FSV}(v) = \emptyset$.

Because of the relative difficulty of formalizing context operations like FV in Twelf, this proof has only been done on paper. Like Property 2, the proof was done for a similar language without a validity judgment or ∂ modality.

6.5 Backend

The backend of the compiler transforms the low-level compiled program into typed assembly language that can be executed by the machine. For the ML5 compiler, I will target the TALT system in active development by Crary *et al.* at Carnegie Mellon [13].

I have not yet designed the backend. Vanderwaart's resource-bounded Popcorn compiler [44] has a code generator that produces TALT code from a low-level typed intermediate language. I plan to try to reuse this code with minor modifications. Even if this turns out to be impossible, generation of typed assembly language is a solved problem; there are only a few new challenges in this setting:

Marshalling. ML5 programs must be able to marshal arbitrary code and data into raw bytes for transmission on the network. By using a dictionary-passing scheme [48] we can arrange for marshalling and unmarshalling functions to be available whenever they are needed. Writing these marshalling functions in a type-safe way will take special care, but the problem has been studied before [15, 14].

Binding to Local Resources. More interesting will be the problem of identifying and binding to local resources. One of the primary motivations of the work is the controlled access to local resources (for example, a value `lineprinter : string` $\rightarrow$ `unit@`$\mathbf{w}_{8108}$). During unmarshalling, code that makes use of a local resource must have these open references bound to those local resources. The problem is slightly different from marshalling, because in general, the host that prepares the marshalled code can only name those resources—it cannot make sense of them in the same way it can make sense of common data like tuples and portable code.

I solved a very similar problem in the Hemlock compiler with an easy ad hoc solution. Generalizing that technique to this case would mean providing a uniform table on each host with one entry for every local resource that is used in the program. The table is initialized by the runtime. On the host where the resource is present, the table entry contains a tagged reference to the resource, and accesses to it succeed. Accesses to non-existent resources fail with a run-time error. Although this scheme would work for the ML5 implementation as well, it is worth investigating how to avoid this run-time check. However, note that even if the implementation incurs the cost of a run-time check, the type system guarantees that such a check will never fail.

Allocation and Valuability. Allocation conversion [28] is a pass that makes the allocation and initialization of memory explicit. The standard way of allocating the tuple $\langle 830, 1\rangle$, for example, is to generate code such as the following:[4]

```
let x = malloc(8, int * int)
let _ = x[0] := 830
let _ = x[1] := 1
```

Several constructs in the internal language, such as `hold` and `cval`, have a syntactic restriction to values that is necessary for safety. Many values, however, require allocation. This makes those values into effectful expressions, such that those programs can no longer be typed in our language! One solution is to relax the value restriction to one of *valuability*. A “valuable” expression may have effects, but it can safely be evaluated by one world on behalf of another. I have only sketched such a system on paper.

By postponing allocation conversion until after translation into a language like TALT that is not modally typed, we can sidestep this issue completely. Still, it is interesting to figure out what the right thing to do in this case is, even if I do not actually implement it.

7 The ML5 Runtime

Related to the backend of the compiler is the language runtime. The runtime is the software that enables a compiled program to run by mediating between it

[4]The actual code would be more complicated, mostly due to type systems for initialization, but that is not at issue here.

and the environment. In the case of a distributed programming language, this includes forming and maintaining the network on which the program runs.

The ML5 runtime will be similar to the ConCert Conductor [29], and I expect to be able to reuse much of the network code developed for that software. In many ways it will be simpler, because it does not need to support features like automatic failure recovery and peer-to-peer node discovery. However, there are again some special concerns for ML5:

Concurrency. A logical account of concurrency is firmly outside the scope of this thesis. However, some minimal support for concurrency will likely be necessary to develop a compelling distributed application. A simple fork/join mechanism like futures (as used in Section 5) may suffice. Typed CML-style channels [38] might be easily implementable using the same marshalling mechanisms already planned.

Any of these should integrate cleanly with the spatial distribution features of ML5, the implementation burden being the main cost. The demands of the application should dictate which features I include.

Garbage Collection. In order to support long-running programs, the runtime must reclaim abandoned storage. The TALT runtime already includes a conservative garbage collector for locally allocated data. However, as references to local data spread (in the environments of mobile code) to other places in the network, we cannot safely reclaim these data without ensuring that all global references have been discarded. Distributed garbage collection is a problem that has been studied, but the solutions are quite complicated [37]. At a minimum, it may be feasible to simply never reclaim data that may have non-local references. If not, then the most complicated scheme I would attempt would be very naïve, stopping the entire world in order to conduct a scan for data with no global references.

8 Application

I plan to demonstrate the effectiveness of ML5 and its implementation by building an example distributed application. The application will be chosen to show off the features that I have time to implement, so I have not committed to a specific application yet. Optimally, the application I choose would have the following characteristics. Most of its complexity should come from the way it is distributed, not from the particular computations it does at each host. It should manage location-dependent resources and effects in a non-trivial way, and should make use of the higher-order programming techniques enabled by ML5's type system. Finally, a similar application should have an implementation in at least one related system, so that the two can be compared.

9 Related Work

Distributed computing is a huge area of computer science, suggesting a large body of related work. Most of this work is only superficially relevant to this proposal, and the relationship can be summarized in terms of a few major themes. A few more closely related languages are described in detail below.

A Language, not a Tool. ML5 is an abstract language designed for spatially distributed computing on a foundation of logic. In contrast, many systems used for distributed computing are ad hoc extensions of existing languages. In many languages used in industry, the mechanisms for distributed computing are provided in the form of libraries and stub generators (*e.g.* CORBA [50, 45], DCOM [7], .Net [33], Web Services [47, 46]).

By building ML5 from first principles, and by using the Curry-Howard Isomorphism to guide its design, we are able to isolate key concepts of spatial distribution and codify them in a natural way within the type system. The result is a language that is more elegant.

Marshalling and Data Mobility. A common shortcoming of distributed languages and tools is that they place restrictions on the types of data that can be marshalled and sent to other hosts. CORBA, whose basic primitive for distributed programming is the remote procedure call, limits the types of RPC arguments to a few simple base types. In Java RMI [16], objects matching the `Serializable` interface can be marshalled, but this does not include references to local resources such as file handles. I contend that this restriction stems from a mistaken conflation of two separate notions. First is the implementation technique of marshalling, which is simply a way of representing a piece of data in a format suitable for transmission on a network. The second is the potential mobility of data, that is, the fact that some data make sense at more than one world. In most languages these ideas are conflated because distributed applications essentially consist of a collection of separate programs, each of which can only understand data from its own perspective. When a Java program performs a remote procedure call, the function that is called can't help but require that all its arguments make sense to it, because the only notion of "making sense" is one of "making sense locally." Therefore, every remote procedure call requires that the arguments shift from making sense to the caller to making sense at the receiver. Because this shift always occurs at the same time as marshalling, they seem like the same operation.

The modal type system of ML5 makes the distinction easy to see, because it endows programs with the ability to simultaneously reason about the well-formedness of values from multiple perspectives. An ML5 program may observe that a value it holds also makes sense at another world, and therefore can coerce it to a value well-typed at that world. If it then performs a remote procedure call, it may pass along that value, knowing it makes sense in the called procedure's world. A program can also pass along a value that only makes sense to it—in which case the called procedure knows that the value it has received is only well-typed at the caller's world. In either case, marshalling is simply an implementation technique; any value can be marshalled for transmission, even if it does not make sense on one end of the transaction. This permits more flexibility in the way programs are written, particularly when programming with higher-order functions.

The Location of Effects. Another feature of ML5's modal typing discipline is that it is able to directly control access to local resources in a statically checked way. I believe that no other distributed programming language features this ability. For instance, in Java RMI almost all checks happen dynamically, which means that more bugs can survive until run-time. These run-time checks include the dynamic serializability of arguments or return types of methods that are invoked remotely, as well as any checks that the Java program must make to ensure that local resources are present when it needs them. For an example of the latter: when implementing a client/server database system, it is possible for the client to instantiate a local instance of the server object

```
Server serv = new Server();
```

instead of getting a remote instance via RMI

```
Server serv =
  (Server)Naming.lookup("rmi://server.org/dbserv");
```

Unless special precautions are taken, this compiles and works until the server code attempts to find the local database on the client computer, which fails. Similarly, the client can call any public helper code or methods on objects that are intended for use only by the server. If the programmer wishes to exclude such erroneous programs, it is often possible to use Java mechanisms like `private` fields and inner classes to control access to location-dependent resources. To do so, one must identify the concept of location with the concept of class membership (the only notion of principal in the language). In contrast, ML5 provides a type structure that allows the programmer to directly express these location dependencies. I claim that this is exactly the relevant concept for spatially distributed programs.

9.1 Distributed ML-Like Languages

Several lines of research have extended ML-like languages with explicit support for distributed computing. Some present a very different abstraction of the network; for instance, D'Caml's model [49] is a single parallel computer with a unified store. This model solves the problem of controlling access to localized resources by making every resource available everywhere through the automatic creation of proxies. This has the disadvantage of obscuring the performance characteristics of the code, because any read, write, or even copy of a memory location can require a request over the network. ML5 is lower level in that it attempts to describe the spatial distribution of data with a type structure rather than abstract away from it.

Acute. Acute [41] is a new ML-like language for distributed computing. The model it supports is more difficult than ours: a dynamic linking setting where interacting programs across hosts may be upgraded to different versions while the program is running. The general mechanism in Acute for controlling locality is its dynamic rebinding system. This allows the programmer to mark libraries so that references to them are rebound to local versions when code arrives at a site. In ML5, every variable or label refers to a single binding site, be it bound in the program or in the initial context that describes the local resources in the network. This improves the programmer's ability to reason statically about programs. On the other hand, dynamic binding is a reality of many distributed applications. It may be possible to explicitly implement dynamic binding on top of ML5's static type system, which would be interesting future work.

Like several other languages in this category, Acute allows for the marshalling of arbitrary values, and because of the dynamic framework must take special care to preserve abstraction between distributed modules and module versions. In comparison, because we check the entire program during a single static link, we are able to treat abstraction as a high-level issue that is simply not present when running machine code.

Alice. Alice [39, 1] is a project to build a new SML-like language with many additional features, including distribution. In Alice, marshalling is called *pickling*, and the story is similar to Java: Stateful values such as arrays are freshly copied, and pickling fails at runtime for "sited" values (those that contain site-specific resources like file handles or thread IDs). As in Java, sitedness is not evident from the type of a value. Hosts have the ability to publish picklable values for receipt by other hosts, and to spawn new threads of control. As usual, there is no typing support of the kind we offer, in fact, "you must be careful not to use sited values in a service!" [2] or run-time errors may occur.

9.2 Modal Logic in Distributed Computing

Other lines of research have used modal logic (in various forms) for distributed computing. Borghuis and Feijs's Modal Type System for Networks [6] provides a logic and operational semantics[5] for network tasks with stationary services and mobile data. They use $\Box$, annotated with a location, to represent services. For example, $\Box^o(A \supset B)$ means a function from A to B at the location o. With no way of internalizing mobility as a proposition, the calculus limits mobile data to base types. Services are similarly restricted to depth-one arrow types. By using $\Box$ for mobile code and $\Diamond$ or `at` for stationary resources, we believe our resulting calculus is both simpler and more general.

[5]By way of compilation into shell scripts!

Cardelli and Gordon [10] provide an early example of using modal logic for reasoning about programs spatially, later refined by Caires and Cardelli [8, 9]. They do not take a propositions-as-types view of their logic; instead, they start from a process calculus, mobile ambients [10], and develop a classical logic for reasoning about their behaviors. Therefore, their modal logic is very different from intuitionistic S5 and includes connectives for stating temporal properties, security properties, and properties of parallel compositions. In contrast, the proposed work gives a computational interpretation of the logic as a type system in the tradition of ML.

Hennessy et al. [19] developed a distributed version of the π-calculus and impose a complex static type system in order to constrain and describe behavior. Similarly, Schmitt and Stefani [40] develop a distributed, higher-order version of the Join Calculus with a complex behavioral type system. In comparison, our system is much simpler, eliminating the complexities of concurrency, access control, and related considerations. By basing our system on the Curry-Howard correspondence, we have a purely logical analysis and, furthermore, propose a straightforward path for integration into a full-scale functional language for realistic programs.

Moody [26] gives a system based on the constructive modal logic S4 due to Pfenning and Davies [36]. In contrast to our explicit world formalism, this language is based on judgments A true (here), A poss (somewhere), and A valid (everywhere). The operational semantics of his system takes the form of a process calculus with nondeterminism, concurrency and synchronization; a significantly different approach from our sequential model. From the standpoint of a multiple world semantics, the accessibility relation of S4 satisfies only reflexivity and transitivity, not symmetry. From the computational point of view, accessibility in his work describes process interdependence rather than connections between actual network locations. Programs are therefore somewhat higher-level and express *potential mobility* instead of explicit code motion as in the `get` construct. In particular, due to the lack of symmetry it is not possible to go back to a source world after a potentially remote procedure call except by returning a value.

Jia and Walker [21] give a judgmental account of an explicit world S5-based logic that is very similar to ours. The main difference is in the envisioned operational semantics. Theirs is a process calculus admitting "action-at-a-distance," where a primitive notion is the passive synchronization of two processes: one that waits for the value of a label, and one that computes it. In comparison, we have designed a semantics that we believe is more amenable to standard functional programming style. Our decomposition of the rules for the modalities means that we have only one primitive that induces communication between nodes, and it has a simple, active, operational behavior. To our knowledge, there is no implementation of their calculus as a programming language.

10 Conclusion

I have presented a plan for the design and construction of a language for spatially distributed computing. The language is designed from first principles on a foundation of logic, using the modal logic Intuitionistic S5 for its ability to represent spatial reasoning. Through its modal type system, ML5 is able to statically control access to location-sensitive resources. The proposed work is novel, feasible, and has both theory and practice components. I conclude with a rough timeline for the completion of the thesis.

Timeline. This project consists of a substantial body of theory (a new language design and associated logics), and implementation (a compiler and distributed runtime). For the purposes of the thesis, these parts may be of varying quality. For instance, a high-performance ML compiler alone can require dozens of man-years of work, but is not required to support the thesis. For the work to be done, I adopt a taxonomy that prioritizes tasks into the following categories:

- **Primary**, meaning that this is a central contribution of original work, deserving maximal attention
- **Secondary**, meaning an interesting and original contribution that is not necessary, but that would strengthen the thesis. Many of these items will become "future work"
- **Tarpit**, meaning a notoriously difficult or involved problem, and so I will only do the bare minimum to make the system usable, leaving the problem open for "future work"
- **Solved**, meaning a task that is well-understood and standard, and so I will do only the bare minimum to make the system usable (or re-use existing technology)

The following is a plan for completing the proposed work, annotated with a basic strategy according to the above. Rather than give time estimates, I give approximate fractions of the total time.

Theory: Language and Logic Design	
Complete Twelf proofs and add other types. *Primary*. What needs to be done here is to work the validity judgment through the phases of the formalized compiler (elaboration, CPS conversion). Since I have worked these cases out on paper, I do not foresee any problems. Adding sums and recursive types (etc.) to the language and proofs will be standard.	6%

Network signatures and separate compilation. *Secondary*. The language as presented does not provide a way of specifying the initial configuration of resources in the network. The language should be extended to give a way to describe the local resources at other hosts, and to compile and link against this interface to them. My recent work on a separate compilation system for Standard ML [43] should help provide guidance in designing this.	6%?
Implementation: Compiler	
Parser, elaboration, CPS, closure conversion. *Primary*. I can re-use some of the code from Hemlock (for instance, the pattern matcher) here.	19%
Standard IL Optimizations. *Solved*. I will spend a small amount of time implementing the optimizations that make the biggest difference for the least effort (dead code elimination), but this is only to make the compiler suitable for reasonable applications.	2%
Optimizations of mobile primitives. *Secondary*. It may be important to do special optimizations for our domain (closure representation, dead and constant `gets`, etc.). Time permitting, I would like to work on this, since it is likely that there is much low-hanging fruit.	2%?
Certifying back-end. *Solved*. I would like to re-use Joe Vanderwaart's popcorn TALT compiler as the back-end, if possible. Although I will need to make some changes to support primitives necessary for ML5-specific features (marshalling), I hope that these will be relatively minor, and that the bulk of boilerplate code will be taken care of.	13%
Implementation: Runtime	

Basic Infrastructure. 13%

Primary. Some basic infrastructure is required to run ML5 programs. This would allow a program to arrive on the target machine, be certified, and executed. Although we cannot use the ConCert v2 system for this, much of that code can be repurposed to build the ML5 runtime. (This runtime is also much simpler, barring the points below, because it does not need to implement work-stealing, P2P node discovery, a distributed hash table, or fault tolerance.) Still, this will be a significant implementation effort.

Concurrency support. ~~∞%~~

Tarpit. Good support for concurrency opens up the system to a large number of design and implementation issues which cannot reasonably fit within the time frame, and which are anyway outside the scope of the thesis. Some basic concurrency support may however be necessary to write a compelling example application.

Failure tolerance. ~~∞%~~

Tarpit. We have designed a fault-tolerant network with the ConCert v2 system. However, the ideas are largely incompatible here because we support effects; fault tolerance would need to be able to log, roll-back, and replay effects. While there are some known techniques for this (such as message logging [22]), they would make the implementation substantially more complex than it needs to be to support the thesis.[6]

Garbage collection. 6%

Tarpit. This is a difficult problem, but a basic solution will be necessary to make the system usable. I will begin with the simplest possible collector and, when designing the runtime, make sure not to preclude more advanced collectors should the need for them arise.

Application

[6]Some very simple support for enabling application authors to handle failure may be possible by simply raising a handleable exception when a `get` does not succeed?

Application. *Primary*. The primary difficulty will be finding an interesting problem to solve that does not require sophisticated concurrency or fault tolerance, since I want to avoid those.	6%
Dissertation	
Writing.	26%

References

[1] Alice web site, 2005. URL: http://www.ps.uni-sb.de/alice/.

[2] Alice web site: Distribution, 2005. URL: http://www.ps.uni-sb.de/alice/manual/distribution.html.

[3] David P. Anderson. BOINC: A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, Pittsburgh, USA, November 2004.

[4] Andrew Appel. *Compiling With Continuations*. Cambridge University Press, Cambridge, 1992.

[5] Robert D. Blumofe and Philip A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *USENIX 1997 Annual Technical Conference on UNIX and Advanced Computing Systems*, pages 133–147, Anaheim, California, 1997.

[6] Tijn Borghuis and Loe M. G. Feijs. A constructive logic for services and information flow in computer networks. *The Computer Journal*, 43(4):274–289, 2000.

[7] D. Box. *Understanding COM*. Addison-Wesley, Reading, MA, 1997.

[8] Luís Caires and Luca Cardelli. A spatial logic for concurrency (part I). In *Theoretical Aspects of Computer Software (TACS)*, pages 1–37. Springer-Verlag LNCS 2215, October 2001.

[9] Luís Caires and Luca Cardelli. A spatial logic for concurrency (part II). In *Proceedings of the 13th International Conference on Concurrency Theory (CONCUR)*, pages 209–225, Brno, Czech Republic, August 2002. Springer-Verlag LNCS 2421.

[10] Luca Cardelli and Andrew D. Gordon. Anytime, anywhere. modal logics for mobile ambients. In *Proceedings of the 27th Symposium on Principles of Programming Languages (POPL)*, pages 365–377. ACM Press, 2000.

[11] B. Chang, K. Crary, M. DeLap, R. Harper, J. Liszka, T. Murphy VII, and F. Pfenning. Trustless grid computing in ConCert. In M. Parashar, editor, *Grid Computing – Grid 2002 Third International Workshop*, pages 112–125, Berlin, November 2002. Springer-Verlag.

[12] ConCert project home page: Software, 2006. URL: `http://msdn.microsoft.com/netframework/`.

[13] Karl Crary. Toward a foundational typed assembly language (expanded version). Technical Report CMU-CS-02-196, Carnegie Mellon University, 2002.

[14] Karl Crary, Michael Hicks, and Stephanie Weirich. Safe and flexible dynamic linking of native code. In *2000 ACM SIGPLAN Workshop on Types in Compilation*, September 2000.

[15] Karl Crary and Stephanie Weirich. Flexible type analysis. In *International Conference on Functional Programming*, pages 233–248, 1999.

[16] Jim Farley. *Java Distributed Computing*. O'Reilly, January 1998.

[17] Ian Foster and Carl Kesselman. The Globus project: a status report. *Future Generation Computer Systems*, 15(5–6):607–621, 1999.

[18] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, San Francisco, California, 1999.

[19] Matthew Hennessy, Julian Rathke, and Nobuko Yoshida. SafeDPi: A language for controlling mobile code. Report 02/2003, Department of Computer Science, University of Sussex, October 2003.

[20] Hybrid logics bibliography, 2005. URL: http://hylo.loria.fr/content/papers.php.

[21] Limin Jia and David Walker. Modal proofs as distributed programs (extended abstract). *European Symposium on Programming*, 2004.

[22] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proc. 7th Annual ACM Symp. on Principles of Distributed Computing*, pages 171–181, Toronto (Canada), 1988.

[23] Clarence Irving Lewis. *A Survey of Symbolic Logic*. University of California Press, 1918.

[24] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997.

[25] Modal types for mobile code: Supplementary twelf source, 2006. URL: http://tom7.org/proposal/.

[26] Jonathan Moody. Modal logic as a basis for distributed computation. Technical Report CMU-CS-03-194, Carnegie Mellon University, October 2003.

[27] Greg Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, December 1995. Available as CMU Technical Report CMU–CS–95–226.

[28] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[29] Tom Murphy, VII. Functional grid programming with ConCert, 2004. Unpublished draft; `http://tom7.org/papers/`.

[30] 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.

[31] 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, July 2004.

[32] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, January 1997.

[33] .Net web site, 2006. URL: `http://msdn.microsoft.com/netframework/`.

[34] Sungwoo Park. A modal language for the safety of mobile values. Technical Report CMU-CS-05-124, Carnegie Mellon, Pittsburgh, Pennsylvania, USA, May 2005.

[35] Leaf Petersen. *Certifying Compilation for Standard ML in a Type Analysis Framework*. PhD thesis, Carnegie Mellon University, May 2005. Available as CMU Technical Report CMU–CS–05–135.

[36] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.

[37] David Plainfossé and Marc Shapiro. A survey of distributed collection techniques. Technical report, BROADCAST, 1994.

[38] John Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[39] Andreas Rossberg, Didier Le Botlan, Guido Tack, Thorsten Brunklaus, and Gert Smolka. *Alice Through the Looking Glass*, volume 5 of *Trends in Functional Programming*. Intellect, Munich, Germany, 2004.

[40] Alan Schmitt and Jean-Bernard Stefani. The M-calculus: A higher-order distributed process calculus. In *Conference Record of the 30th Symposium on Principles of progr ammming Languages*, pages 50–61, New Orleans, Louisiana, January 2003. ACM Press.

[41] Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute: high-level programming language design for distributed computation. In *International Conference on Functional Programming (ICFP) 2005*, Tallinn, Estonia, September 2005.

[42] Alex Simpson. *The Proof Theory and Semantics of Intuitionistic Modal Logic*. PhD thesis, University of Edinburgh, 1994.

[43] David Swasey, Tom Murphy, VII, Karl Crary, and Robert Harper. A separate compilation extension to Standard ML. Technical Report CMU-CS-06-104, Carnegie Mellon University, January 2006.

[44] Joseph C. Vanderwaart and Karl Crary. Automated and certified conformance to responsiveness policies. In *Proceedings of the ACM SIGPLAN workshop on types in language design and implementation (TLDI)*, 2005.

[45] Steve Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), 1997.

[46] W3C Web Service Description Working Group, David Booth and Canyang Kevin Liu, eds. Web services description language (WSDL) version 2.0 part 0: primer. Technical Report WD-wsdl20-primer-20050510, W3C, URL: http://www.w3.org/TR/wsdl20-primer, May 2005.

[47] W3C web services working group, 2002. URL: http://www.w3.org/2002/ws/.

[48] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989.

[49] Ken Wakita, Takashi Asano, and Masataka Sassa. D'Caml: A native distributed ML compiler for heterogeneous environment. In Patrick Amestoy, Philippe Berger, Michel J. Daydé, Iain S. Duff, Valérie Frayssé, Luc Giraud, and Daniel Ruiz, editors, *Euro-Par 1999 Parallel Processing, 5th International Euro-Par Conference*, volume 1685 of *Lecture Notes in Computer Science*, pages 914–924. Springer, 1999.

[50] Zhonghua Yang and Keith Duddy. CORBA: A platform for distributed object computing (a state-of-the-art report on OMG/CORBA). *Operating Systems Review*, 30(2):4–31, 1996.

A Intuitionistic S5

(a)

$$\frac{\Gamma, \omega\ \mathsf{world} \vdash M : A@\omega}{\Gamma \vdash \mathsf{box}\ \omega.M : \Box A@\mathrm{w}}\ \Box\,\mathrm{I} \qquad \frac{\Gamma \vdash M : \Box A@\mathrm{w}}{\Gamma \vdash \mathsf{unbox}\ M : A@\mathrm{w}}\ \Box\,\mathrm{E}$$

$$\frac{\Gamma \vdash A\ \mathsf{true}@\mathrm{w}'}{\Gamma \vdash \Diamond A\ \mathsf{true}@\mathrm{w}}\ \Diamond\,\mathrm{I} \qquad \frac{\begin{array}{l}\Gamma \vdash \Diamond A\ \mathsf{true}@\mathrm{w}' \\ \Gamma, \omega\ \mathsf{world}, A\ \mathsf{true}@\omega \vdash C\ \mathsf{true}@\mathrm{w}\end{array}}{\Gamma \vdash C\ \mathsf{true}@\mathrm{w}}\ \Diamond\,\mathrm{E}$$

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

$$\frac{\Gamma \vdash A\ \mathsf{true}@\mathrm{w} \quad \Gamma \vdash B\ \mathsf{true}@\mathrm{w}}{\Gamma \vdash A \wedge B\ \mathsf{true}@\mathrm{w}}\ \wedge\mathrm{I} \qquad \frac{\Gamma \vdash A \wedge B\ \mathsf{true}@\mathrm{w}}{\Gamma \vdash A\ \mathsf{true}@\mathrm{w}}\ \wedge\mathrm{E}_1$$

$$\frac{\Gamma \vdash A \wedge B\ \mathsf{true}@\mathrm{w}}{\Gamma \vdash B\ \mathsf{true}@\mathrm{w}}\ \wedge\mathrm{E}_2 \qquad \frac{\Gamma \vdash A\ \mathsf{true}@\mathrm{w}}{\Gamma \vdash A \vee B\ \mathsf{true}@\mathrm{w}}\ \vee\mathrm{I}_1$$

$$\frac{\Gamma \vdash B\ \mathsf{true}@\mathrm{w}}{\Gamma \vdash A \vee B\ \mathsf{true}@\mathrm{w}}\ \vee\mathrm{I}_2 \qquad \frac{\begin{array}{l}\Gamma \vdash A \vee B\ \mathsf{true}@\mathrm{w}' \\ \Gamma, A\ \mathsf{true}@\mathrm{w}' \vdash C@\mathrm{w} \\ \Gamma, B\ \mathsf{true}@\mathrm{w}' \vdash C@\mathrm{w}\end{array}}{\Gamma \vdash C\ \mathsf{true}@\mathrm{w}}\ \vee\mathrm{E}$$

(b)

$$\frac{\Gamma \vdash A\ \mathsf{true}@\mathrm{w}'}{\Gamma \vdash A\ \mathsf{at}\ \mathrm{w}'\ \mathsf{true}@\mathrm{w}}\ \mathtt{at}\,\mathrm{I} \qquad \frac{\Gamma \vdash A\ \mathsf{at}\ \mathrm{w}\ \mathsf{true}@\mathrm{w}'}{\Gamma \vdash A\ \mathsf{true}@\mathrm{w}}\ \mathtt{at}\,\mathrm{E}$$

Figure 6: (a) The standard presentation of Intuitionistic S5, as used by *e.g.* Simpson [42]. (b) The extension to add the `at` modality, as in *e.g.* Jia and Walker [21]

B Internal Language

$$
\begin{array}{rccl}
\text{exps} & M, N & ::= & \texttt{leta}\ x = M\ \texttt{in}\ N \\
 & & | & v \\
 & & | & M N \\
 & & | & \texttt{fst}\ M \mid \texttt{snd}\ M \\
 & & | & M[\mathrm{w}] \\
 & & | & \texttt{unpack}\ M\ \texttt{as}\ \omega, x\ \texttt{in}\ N \\
 & & | & \texttt{get}[\mathrm{w}; M]N \\
 & & | & \texttt{localhost}() \\
 & & | & \texttt{letc}\ u = M\ \texttt{in}\ N \\
 & & | & \texttt{mkpair}(M, N) \\
\text{valid exps} & S & ::= & \texttt{mobc}\ M \\
 & & | & \texttt{valid}\ z \\
\text{values} & v & ::= & \langle v_1, v_2 \rangle \\
 & & | & \texttt{hold}_{\mathrm{w}} v \\
 & & | & \lambda x.M \\
 & & | & \Lambda \omega.v \\
 & & | & \overline{\mathbf{w}} \\
 & & | & u \mid x \mid z \\
 & & | & () \\
 & & | & \texttt{pack}\ \mathrm{w}, v\ \texttt{as}\ \exists \omega.A \\
 & & | & \texttt{cir}\ S \\
\text{valid vals} & z & ::= & \texttt{vv}\ \omega.v \\
\text{types} & A, B & ::= & A \times B \mid A \to B \\
 & & | & \forall \omega.A \mid \exists \omega.A \\
 & & | & \mathrm{w}\ \texttt{addr} \mid \texttt{unit} \\
 & & | & \bigcirc A
\end{array}
$$

Figure 7: The syntax of the Internal Language as formalized in the Twelf code. In this version, we use a separate syntactic class z for valid values. These are instantiated at the current world when used as values

For the full static semantics for machine states and the statement of type safety, see the Twelf code [25].

$$
\begin{array}{rrcl}
\text{stack frames} & f & ::= & \texttt{get}[\mathrm{w}; \circ]\ \texttt{M} \\
 & & | & \texttt{ret}[\mathbf{w}]\ \circ \\
 & & | & \circ\, N \mid v\, \circ \\
 & & | & \circ[\mathrm{w}] \\
 & & | & \texttt{fst}\ \circ \mid \texttt{snd}\, \circ \\
 & & | & \texttt{mkpair}(\circ, N) \mid \texttt{mkpair}(v, \circ) \\
 & & | & \texttt{leta}\, x = \circ\ \texttt{in}\ N \\
 & & | & \texttt{unpack}\ \circ\ \texttt{as}\ \omega, x\ \texttt{in}\ N \\
 & & | & \texttt{letc}\, u = \circ\ \texttt{in}\ N \\
 & & | & \texttt{letc\_mob}\, u = \circ\ \texttt{in}\ N \\
\text{stacks} & s & ::= & \texttt{finish} \mid s \lhd f
\end{array}
$$

Figure 8: Stack syntax for the internal language, used to define the dynamic semantics

$$
\begin{array}{rcl}
[\![\texttt{hold}_{\mathrm{w}} v]\!]_{\mathsf{MV}} & = & \texttt{vv}\ \omega.\texttt{hold}_{\mathrm{w}} v \\
[\![\langle v_1, v_2\rangle]\!]_{\mathsf{MV}} & = & \texttt{vv}\ \omega.\langle [\![v_1]\!]_{\mathsf{MV}}, [\![v_2]\!]_{\mathsf{MV}}\rangle \\
[\![\Lambda\omega.v]\!]_{\mathsf{MV}} & = & \texttt{vv}\ \omega'.\Lambda\omega.[\![v]\!]_{\mathsf{MV}} \\
[\![\texttt{pack}\ \mathrm{w}, v\ \texttt{as}\ \exists\omega.A]\!]_{\mathsf{MV}} & = & \texttt{vv}\ \omega'.\texttt{pack}\ \mathrm{w}, [\![v]\!]_{\mathsf{MV}}\ \texttt{as}\ \exists\omega.A \\
[\![\overline{\mathbf{w}}]\!]_{\mathsf{MV}} & = & \texttt{vv}\ \omega.\overline{\mathbf{w}} \\
[\![z]\!]_{\mathsf{MV}} & = & z
\end{array}
$$

Figure 9: Mobilization of values in the internal language. Values of certain forms can be transformed into valid values; the partial function $[\![\cdot]\!]_{\mathsf{MV}}$ performs this mobilization. In each case, the bound world is unused in the body of the valid value

$$
\begin{array}{lcl}
\mathbf{w}{::}s; \texttt{localhost} & \mapsto & \mathbf{w}{::}s; \overline{\mathbf{w}} \\
\mathbf{w}{::}s; \texttt{unpack}\ M\ \texttt{as}\ \omega, x\ \texttt{in}\ N & \mapsto & \mathbf{w}{::}s \lhd \texttt{unpack}\ \circ\ \texttt{as}\ \omega, x\ \texttt{in}\ N; M \\
\mathbf{w}{::}s \lhd \texttt{unpack}\ \circ\ \texttt{as}\ \omega, x\ \texttt{in}\ N; & \mapsto & \\
\qquad\qquad \texttt{pack}\ \mathrm{w}, v\ \texttt{as}\ \exists\omega.A & & \mathbf{w}{::}s; [\,{}^{v}\!/_{x}]N \\
\mathbf{w}{::}s; \texttt{fst}\ M & \mapsto & \mathbf{w}{::}s \lhd \texttt{fst}\ \circ; M \\
\mathbf{w}{::}s; \texttt{snd}\ M & \mapsto & \mathbf{w}{::}s \lhd \texttt{snd}\ \circ; M \\
\mathbf{w}{::}s \lhd \texttt{fst}\ \circ; \langle v_1, v_2 \rangle & \mapsto & \mathbf{w}{::}s; v_1 \\
\mathbf{w}{::}s \lhd \texttt{snd}\ \circ; \langle v_1, v_2 \rangle & \mapsto & \mathbf{w}{::}s; v_2 \\
\mathbf{w}{::}s; M N & \mapsto & \mathbf{w}{::}s \lhd \circ\ N; M \\
\mathbf{w}{::}s \lhd \circ\ N; \lambda x.M & \mapsto & \mathbf{w}{::}s \lhd \lambda x.M\ \circ; N \\
\mathbf{w}{::}s \lhd \lambda x.M\ \circ; v & \mapsto & \mathbf{w}{::}s; [\,{}^{v}\!/_{x}]M \\
\mathbf{w}{::}s; M[\mathrm{w}] & \mapsto & \mathbf{w}{::}s \lhd \circ[\mathrm{w}]; M \\
\mathbf{w}{::}s \lhd \circ[\mathrm{w}]; \Lambda\omega.v & \mapsto & \mathbf{w}{::}s; [\,{}^{\mathrm{w}}\!/_{\omega}]v \\
\mathbf{w}{::}s; \texttt{get}[\mathrm{w}, M]N & \mapsto & \mathbf{w}{::}s \lhd \texttt{get}[\mathrm{w}, \circ]N; M \\
\mathbf{w}{::}s \lhd \texttt{get}[\mathrm{w}, \circ]N; \overline{\mathbf{w}'} & \mapsto & \mathbf{w}'{::}s \lhd \texttt{ret}[\mathbf{w}]\ \circ; N \\
\mathbf{w}{::}s \lhd \texttt{ret}[\mathbf{w}']\ \circ; v & \mapsto & \mathbf{w}'{::}s; [\![v]\!]_{\mathsf{MV}} \\
\mathbf{w}{::}s; \texttt{mkpair}(M, N) & \mapsto & \mathbf{w}{::}s \lhd \texttt{mkpair}(\circ, N); M \\
\mathbf{w}{::}s \lhd \texttt{mkpair}(\circ, N); v & \mapsto & \mathbf{w}{::}s \lhd \texttt{mkpair}(v, \circ); N \\
\mathbf{w}{::}s \lhd \texttt{mkpair}(v_1, \circ); v_2 & \mapsto & \mathbf{w}{::}s; \langle v_1, v_2 \rangle \\
\mathbf{w}{::}s; \texttt{letc}\ u = \texttt{valid}\ z\ \texttt{in}\ N & \mapsto & \mathbf{w}{::}s; [\,{}^{z}\!/_{u}]N \\
\mathbf{w}{::}s; \texttt{letc}\ u = \texttt{mobc}\ M\ \texttt{in}\ N; & \mapsto & \mathbf{w}{::}s \lhd \texttt{letc\_mob}\ u = \circ\ \texttt{in}\ N; M \\
\mathbf{w}{::}s \lhd \texttt{letc\_mob}\ u = \circ\ \texttt{in}\ N; v & \mapsto & \mathbf{w}{::}s; [\,{}^{[\![v]\!]_{\mathsf{MV}}}\!/_{u}]N \\
\mathbf{w}{::}s; \texttt{leta}\ x = M\ \texttt{in}\ N & \mapsto & \mathbf{w}{::}s \lhd \texttt{leta}\ x = \circ\ \texttt{in}\ N; M \\
\mathbf{w}{::}s \lhd \texttt{leta}\ x = \circ\ \texttt{in}\ N; \texttt{hold}_{\mathrm{w}} v & \mapsto & \mathbf{w}{::}s; [\,{}^{v}\!/_{x}]N \\
\mathbf{w}{::}s; \texttt{vv}\ \omega.v & \mapsto & \mathbf{w}{::}s; [\,{}^{\mathbf{w}}\!/_{\omega}]v \\
\mathbf{w}{::}\texttt{finish}; v & \mapsto & \mathbf{w}{::}\texttt{finish}; v
\end{array}
$$

Figure 10: Dynamic semantics for the internal language, given in terms of a relation $\mapsto$ between machine states. A machine state takes the form $\mathbf{w}{::}s; M$ or $\mathbf{w}{::}s; v$ where $\mathbf{w}$ is the current world of evaluation, s is the current stack, and M is the expression to be evaluated or v the value to be returned. The final rule steps from a finished machine to itself, which simplifies the statement of type safety

C CPS Language

$$\begin{array}{rcll}
[\![{}^{\omega.v}/_{u}]\!]_{\mathrm{w}}\, \mathtt{lift}\, u' = v'\ \mathtt{in}\ c & = & \mathtt{lift}\, u' = [\![{}^{\omega.v}/_{u}]\!]_{\mathrm{w}} v'\ \mathtt{in}\ [\![{}^{\omega.v}/_{u}]\!]_{\mathrm{w}} c & \\
[\![{}^{\omega.v}/_{u}]\!]_{\mathrm{w}} \mathtt{halt} & = & \mathtt{halt} & \\
[\![{}^{\omega.v}/_{u}]\!]_{\mathrm{w}} \mathtt{go}[\mathrm{w}'; v_a]\ c & = & \mathtt{go}[\mathrm{w}'; [\![{}^{\omega.v}/_{u}]\!]_{\mathrm{w}} v_a]\ [\![{}^{\omega.v}/_{u}]\!]_{\mathrm{w}'} c & \\
 & \ldots & & \\
 & & & \\
[\![{}^{\omega.v}/_{u}]\!]_{\mathrm{w}} u & = & [{}^{\mathrm{w}}/_{\omega}]v & \\
[\![{}^{\omega.v}/_{u}]\!]_{\mathrm{w}} u' & = & u' & (u \neq u') \\
[\![{}^{\omega.v}/_{u}]\!]_{\mathrm{w}} \lambda x.c & = & \lambda x.[\![{}^{\omega.v}/_{u}]\!]_{\mathrm{w}} c & \\
[\![{}^{\omega.v}/_{u}]\!]_{\mathrm{w}} x & = & x & \\
[\![{}^{\omega.v}/_{u}]\!]_{\mathrm{w}} \overline{\mathbf{w}} & = & \overline{\mathbf{w}} & \\
[\![{}^{\omega.v}/_{u}]\!]_{\mathrm{w}} \mathtt{hold}_{\mathrm{w}'} v' & = & \mathtt{hold}_{\mathrm{w}'} [\![{}^{\omega.v}/_{u}]\!]_{\mathrm{w}'} v' & \\
[\![{}^{\omega.v}/_{u}]\!]_{\mathrm{w}} \mathtt{cval}\ \omega'.v' & = & \mathtt{cval}\ \omega'.[\![{}^{\omega.v}/_{u}]\!]_{\omega'} v' & \\
 & \ldots & &
\end{array}$$

Figure 11: Instantiating substitution for valid variables in the the CPS language. This represents one of two ways used to formulate the dynamic semantics for valid variables. (The other, used exclusively in the Twelf formalization, can be seen in the semantics for the Internal Language in Figure 10.) In this case, the instantiating substitution $[\![{}^{\omega.v}/_{u}]\!]_{\mathrm{w}} c$ substitutes for the variable u within c, by instantiating the valid value $\omega.v$ at the worlds where it is used. The parameter w indicates the current world in the substitution function. Substitution is also defined over values. Note that the instantiation is purely static; in an implementation, the same exact value can be used at each occurrence of u. As usual, we assume the freshness of bound variables in order to avoid capture; the variable u' in the case for `lift`, for instance, is assumed to not appear in v

Property 4 (Type Safety of CPS Language)

(a) *If* $\cdot \vdash c@\boldsymbol{w}$ *(in CPS)*
then there exists $\boldsymbol{w}'$, c' *such that*
$\boldsymbol{w}::c \mapsto \boldsymbol{w}'::c'$.

(b) *If* $\cdot \vdash c@\boldsymbol{w}$ *(in CPS)*
and $\boldsymbol{w}::c \mapsto \boldsymbol{w}'::c'$
then $\cdot \vdash c'@\boldsymbol{w}'$.

$$
\begin{array}{rcl}
\mathbf{w}{::}\texttt{leta}\, x = \texttt{hold}_{\mathrm{w}} v \ \texttt{in}\ c & \mapsto & \mathbf{w}{::}[\,{}^{v}\!/_{x}]c \\
\mathbf{w}{::}\ \texttt{letg}\, u = \texttt{cval}\ \omega'.v \ \texttt{in}\ c & \mapsto & \mathbf{w}{::}[\![\,{}^{\omega'.v}\!/_{u}]\!]_{\mathbf{w}} c \\
\mathbf{w}{::}\ \texttt{lift}\, u = v \ \texttt{in}\ c & \mapsto & \mathbf{w}{::}[\,{}^{v}\!/_{u}]c \\
\mathbf{w}{::}\texttt{let}\ x = \texttt{fst}\ \langle v_1, v_2\rangle \ \texttt{in}\ c & \mapsto & \mathbf{w}{::}[\,{}^{v_1}\!/_{x}]c \\
\mathbf{w}{::}\texttt{let}\ x = \texttt{snd}\ \langle v_1, v_2\rangle \ \texttt{in}\ c & \mapsto & \mathbf{w}{::}[\,{}^{v_2}\!/_{x}]c \\
\mathbf{w}{::}\texttt{let}\ x = \texttt{mkpair}\ v_1, v_2 \ \texttt{in}\ c & \mapsto & \mathbf{w}{::}[\,{}^{\langle v_1, v_2\rangle}\!/_{x}]c \\
\mathbf{w}{::}\texttt{let}\ x = \texttt{localhost}() \ \texttt{in}\ c & \mapsto & \mathbf{w}{::}[\,{}^{\overline{\mathbf{w}}}\!/_{x}]c \\
\mathbf{w}{::}\texttt{go}[\mathbf{w}_1; \overline{\mathbf{w}_1}]\ c & \mapsto & \mathbf{w}_1{::}c \\
\mathbf{w}{::}\texttt{call}\ (\lambda x.c)(v) & \mapsto & \mathbf{w}{::}[\,{}^{v}\!/_{x}]c \\
\mathbf{w}{::}\texttt{halt} & \mapsto & \mathbf{w}{::}\texttt{halt}
\end{array}
$$

Figure 12: The dynamic semantics for the CPS language. It is given in terms of a stepping relation $\mapsto$ between pairs of CPS machines of the form $\mathbf{w}{::}c$, where $\mathbf{w}$ is the current world of evaluation and c is the current continuation expression. We give only the rules for the constructs that appear in Figure 3. The expression `halt` has a self-transition for the ease of stating type safety (Property 4)