Carnegie Mellon University
Research Showcase @ CMU

Computer Science Department | School of Computer Science

7-2010

A Monadic Formalization of ML5

Daniel R. Licata
Carnegie Mellon University

Robert Harper
Carnegie Mellon University

Follow this and additional works at: http://repository.cmu.edu/compsci

This Conference Proceeding is brought to you for free and open access by the School of Computer Science at Research Showcase @ CMU. It has been accepted for inclusion in Computer Science Department by an authorized administrator of Research Showcase @ CMU. For more information, please contact research-showcase@andrew.cmu.edu.

A Monadic Formalization of ML5

Daniel R. Licata* Robert Harper*
Carnegie Mellon University
{drl,rwh}@cs.cmu.edu

ML5 is a programming language for spatially distributed computing, based on a Curry-Howard correspondence with the modal logic S5. However, the ML5 programming language differs from the logic in several ways. In this paper, we give a semantic embedding of ML5 into the dependently typed programming language Agda, which both explains these discrepancies between ML5 and S5 and suggests some simplifications and generalizations of the language. Our embedding translates ML5 into a slightly different logic: intuitionistic S5 extended with a lax modality that encapsulates effectful computations in a monad. Rather than formalizing lax S5 as a proof theory, we *embed* it as a universe within the the dependently typed host language, with the universe elimination given by implementing the modal logic's Kripke semantics.

1 Introduction

One of the many benefits of formalizing programming languages and logics is that the process of formalization, and the constraints of the particular techniques used, can lead to new insights about the system being studied. This paper provides a worked example of this phenomenon, investigating the ML5 programming language for spatially distributed computing [23]. ML5 has previously been formalized [23] using syntactic methods in Twelf [26]. However, we wished to give a semantic interpretation of ML5 into a dependently typed programming language, as a first step towards extending work on embeddings of security typed-languages [22] to account for spatially distributed access control, as in the PCML5 extension of ML5 [7]. Our semantic formalization of ML5 provides insight into several discrepancies between ML5 and the logic upon which it is based, and suggests some simplifications and generalizations of the language, as we now describe.

ML5 facilitates distributed programs that deal with *located resources*, such as a database on a server, the browser display on a client, or heap references on any particular site. When a distributed program running at one site attempts to access a resource located at another site, the program must either communicate with the other site or fail. Because tacit communication makes it very difficult to reason about the execution time of a program (e.g. every memory dereference might involve a network communication), ML5 is based on the stance that all communication should be explicit in the program. However, rather than letting accesses from the wrong site fail dynamically, ML5 employs a type system based on a Curry-Howard correspondence with the modal logic S5 to catch these errors statically. ML5 is defined as an intuitionistic modal logic in the style of Simpson [28], where hypotheses and conclusions are considered relative to *worlds*, which represent places on a network. The ML5 typing judgement has the form $x_1 : A_1[w_1], \ldots, x_n : A_n[w_n] \vdash e : C[w]$, where A_i and C are modal types and w_i and w are worlds.

*This research was sponsored in part by the National Science Foundation under grant number CCF-0702381 and by the Pradeep Sindhu Computer Science Fellowship. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Preliminary Report. Final version to appear in:
Post-Proceedings of the 5th International Workshop on Logical Frameworks and Meta-languages: Theory and Practice, 2010

Despite being designed by a correspondence with S5 modal logic, the ML5 programming language differs from S5 in several ways: First, ML5 ensures that all communication is explicit in the program by providing only a single communication primitive that, operationally, goes to world w', runs e, and brings the resulting value back to w:

$$\frac{\Gamma \vdash e : A[w'] \quad A \text{ mobile}}{\Gamma \vdash \mathsf{get}\ e : A[w]}$$

The condition A mobile rules out instances of get where A is, for example, ref int—which would take a reference that should be used at w' and turn it into a reference that should be used at w, violating the intended typing guarantees.

Second, in the standard presentation of S5, elimination rules for positive connectives such as sums allow an arbitrary conclusion, which is unconnected to the principal formula. In ML5, this rule is *tethered*, in that the world in the conclusion must be equal to the world in the premise:

$$\frac{\begin{array}{l}\Gamma \vdash A \vee B[w] \\ \Gamma, x : A[w] \vdash C[w'] \\ \Gamma, x : B[w] \vdash C[w']\end{array}}{\Gamma \vdash C[w']}\ \text{untethered case} \qquad \frac{\begin{array}{l}\Gamma \vdash A \vee B[w] \\ \Gamma, x : A[w] \vdash C[w] \\ \Gamma, x : B[w] \vdash C[w]\end{array}}{\Gamma \vdash C[w]}\ \text{tethered case}$$

ML5 makes the tethering restriction because the obvious operational interpretation of the untethered rule requires communication (go to `w` to run the principal formula). Indeed, the untethered rule is derivable using `get`. However, this tethering is at odds with the Kripke semantics of modal logic, where $A \vee B[w]$ is interpreted as $A[w]$ or $B[w]$—if the interpretation commutes with disjunction, then a disjunction should be eliminable no matter the conclusion of the sequent.

Third, ML5 includes two different $\Box$-like modalities with the same introduction rule. The first is written $\mathfrak{B}\, A$, while the second, $\forall w.A$ at w, is a composition of the connective $\forall$ (quantification over worlds) and the hybrid logic [5] at modality, which internalizes the judgement $A[w]$ as a connective. (*Hybrid logic* is between modal logic (truth is relativised to worlds) and first-order logic (propositions may mention worlds)). The two connectives are eliminated differently: ML5 distinguishes a syntactic category of values from ordinary expressions, and $\mathfrak{B}\, A$ can be eliminated to construct a value but $\forall w.A$ at w cannot.

In addition to these discrepancies, there is some confusion over the meaning the world in ML5 value judgements $v :: A[w]$ and expression judgements $e : A[w]$. An expression judgement means that e is an expression that must be evaluated at w, and produces a value $v :: A[w]$, but what does the world in the value judgement mean? One cannot think of values as just a subset of expressions, as the value rules for certain connectives, such as at and $\mathfrak{B}$, would violate the property that all communication happens through `get`. Additionally, in the dynamic semantics of the ML5 internal language given in Section 3.3 of Murphy [23], get e returns the entire value of e to the calling world, so the value judgement does not mean that the value v is physically located at w.

In this paper, we propose a new logical foundation for ML5, which explains the differences between ML5 and S5 and clarifies the role of the world in a value judgement. We translate ML5 into the intuitionistic logic S5 extended with a lax modality, written $\bigcirc A$, that encapsulates effectful computations in a monad [10, 15, 21]. This monadic distinction between pure terms and effectful computations is already tacit in ML5's distinction between values and expressions—and in intermediate languages used in the ML5 compiler (e.g. the CPS language in Murphy [23]), which include elimination forms for "values" as "values". Here, we draw out this distinction by formalizing a monadic interpretation of ML5, and make some improvements to the language based on this formulation.

In our interpretation, ML5 values of type $A[w]$ are interpreted as pure terms of type $A^*\langle w\rangle$, where A^* is a monadic translation of A, and we write $B\langle w\rangle$ for a worlded type in the lax modal logic. On the other hand, potentially effectful ML5 expressions are interpreted as inhabitants of type $(\bigcirc A^*)\langle w\rangle$. The role of the world in a value judgement, i.e. the role of the world in $B\langle w\rangle$, is to describe where the resources in subexpressions of the type A may be used and where the computations in the type A must be run. For example, the value judgement $(\mathsf{ref\ int} \supset \bigcirc\mathsf{unit})\langle\mathsf{client}\rangle$ describes a function that takes a reference that must be used at the client and produces a computation that must be run at the client.

Our interpretation explains the three discrepancies between ML5 and S5 mentioned above: the get primitive is an extra operation on the monad $\bigcirc$ that allows a computation that must be run at one world to be run from another. The tethered `case` rule is a derived rule: in ML5, the scrutinee of the case is an effectful expression, so it is necessary to sequence evaluating this expression with an actual case analysis on the value produced—and it is the *sequencing* that requires tethering. Indeed, we show that we can enrich ML5 with an untethered case rule on *values*, which would permit simpler code. Finally, the $\mathfrak{S}$ connective can be eliminated in favor of $\forall$ and `at`, given the standard pure elim rules for these types.

Rather than formalizing lax S5 as a proof theory, we *embed* it inside a dependently typed host language, Agda [25]. First, we define a lax logic for distributed programming, L5, which is embedded in Agda using an indexed monad of computations at a place. Next, we define a universe of hybrid modal types, HL5, and give them meaning by interpretation into L5—i.e. we define a syntax of HL5 types, along with a function interpreting them as L5 types. Finally, we translate ML5 into HL5. This technique saves us the work of defining a proof theory for HL5, and additionally allows us to inherit the equational theory of the meta-language, which can be exploited in proving that the semantics validates the operational semantics of ML5. While it is simple to embed type systems specified by standard judgements of the form $x_1 : A_1, \ldots, x_n : A_n \vdash e : C$ as a universe, it requires a bit of thought to adapt these techniques to languages with modal type systems, such as HL5. In previous work on programming with variable binding [20], we employed a technique for embedding such modal type systems: intuitionistic modal logics can be given a Kripke semantics in first-order intuitionistic logic [28], and we can formalize this semantics in a dependently typed language host language. However, the presentation of this technique in that paper was somewhat obscured by the particular example. In this paper, we present this technique in a simpler setting, and apply it to explain the proof theory of ML5. The Agda code for this paper is available from `www.cs.cmu.edu/~drl`.

We briefly review Agda's syntax; see the Agda Wiki(`wiki.portal.chalmers.se/agda/`) for more introductory materials. Dependent function types are written with parentheses as `(x : A) → B`. An implicit dependent function space is written as `{x : A} → B` or `∀ {x} → B` and arguments to implicit functions are inferred. Non-dependent functions are written as `A → B`. Functions are written as `λ x → e`. Named functions are defined by clausal pattern-matching definitions. `Set` is the classifier of classifiers in Agda, like the kind `type` in ML or Haskell.

2 L5

In this section, we define an interface for distributed programming in Agda, based on lax logic [10, 15]. We define a type of worlds and a family of monads `IO w S` indexed by worlds, which represent an effectful computation that runs at world `w` and produces a value of type `S`.

```
World : Set
server : World
client : World
```

```
IO     : World → Set → Set
return : ∀ {w S} → S → IO w S
_»=_    : ∀ {w S S'} → IO w S → (S → IO w S') → IO w S'
get    : ∀ {w' w S} → IO w' S → IO w S

Ref : World → Set → Set
_:=_ : ∀ {w S} → Ref w S → S → IO w Unit
!    : ∀ {w S} → Ref w S → IO w S
new  : ∀ {w S} → IO w (Ref w S)
```

The definitions are parametrized by a type of worlds, which we here assume to contain client and server. `IO w S` is axiomatized as a monad, with `return` and bind (»=), along with an additional operation `get` that allows the computation to switch to a different world. The command `get` can be thought of as a remote procedure call, which goes to `w'`, runs the given command, and then brings the resulting value back to `w`. Next, we define a type `Ref w S` of heap references located at world `w`. The operations for setting (`:=`), getting (`!`), and creating (`new`) references require the reference to be at the same world as the computation.

Because these types are embedded in Agda, they may interact freely with the host language types—e.g., a computation may return value of any Agda `Set`, and we can use Agda Π and Σ types to quantify over worlds.

In Agda, we use the *postulate* keyword to assume an implementation of this interface. Under the hood, these operations can be implemented using foreign-function calls, e.g. interpreting `IO k A` as the `IO` monad in Haskell. A trivial implementation of this interface may run all computations in a single executable on a single machine. A more realistic implementation would require compiler support for (a) compiling a single program to run on multiple hosts and (b) marshaling and unmarshaling all values (the ML5 compiler implements both of these). We also give a high-level abstract operational semantics for computations in the companion code.

L5 satisfies the design goals of ML5: it ensures that resources are only used by a computation running at the appropriate world, and it also makes all communication explicit via `get`.

The following example illustrates the use of the indexed types `Ref` and `IO`, as well as the `get` operation on the monad:

```
update : (Ref server (IO server Unit))  →  IO client Unit  →  IO client Unit
update l clicomp = get{server} (l := (get{client} clicomp))
```

This function takes a reference at the server that stores a callback computation, which itself runs at the server, as well as a computation that runs on the client, and produces a computation that runs on the client. This computation goes to the server (the outer `get`), and updates the callback ref (`l`) to point to a computation that goes back to the client and runs `clicomp` (the inner `get`). Omitting either `get` causes a type error, preventing resources from being used at the wrong location. In fact, nothing about the code is specific to the worlds `client` and `server`, so we can make it polymorphic in the worlds:

```
update' : (w1 : World) (w2 : World) → Ref w2 (IO w2 Unit) → IO w1 Unit → IO w1 Unit
update' w1 w2 l comp1 = get{w2} (l := (get{w1} comp1))
```

3 HL5

A disadvantage of L5 is that types can be somewhat verbose, as they they repeat the same world multiples times (e.g. both `client` and `server` occur twice in the type of `update`). A hybrid modal type system,

as in ML5, permits more concise specifications: one writes a modal type `A`, which for the most part does not mention worlds, and then interprets it relative to a world at the outside by writing `A < w >`. In this section, we define HL5, which is a hybrid type system constructed on top of the above lax logic. We define HL5 by semantic embedding into L5 (which itself is just a library in Agda): we define a syntax of HL5 types, and then an interpretation function mapping HL5 types to functions from worlds to Agda `Sets` (or, thinking constructively, predicates on worlds).

3.1 HL5 Types

The datatype of HL5 types is defined as follows (note that Agda allows multiple datatype constructors per line):

```
data Type : Set where
  _⊃_ _∨_ : Type → Type → Type
  ∀₅ ∃₅ : (World → Type) → Type
  _at_ : Type → World → Type
  ◯    : Type → Type
  ref  : Type → Type
```

The types $\supset$ and $\vee$ represent functions and sums (the notation `_∨_` allows $\vee$ to be used infix). The types $\forall_5$ and $\exists_5$ represent quantifiers over worlds. Next, `at` is a connective of hybrid logic, which allows types to set the world at which the type is interpreted. Finally, $\bigcirc$ and `ref` represent monadic computations and references; note that they are *not* indexed by a world. Box and diamond can be defined using quantifiers and `at`: `□ A = ∀₅ (λ w → A at w)` and `◇ A = ∃₅ (λ w → A at w)`.

Below we define the interpretation function `A < w >`, which takes a modal type and a world and produces an Agda `Set`. Using modal types, we can rewrite the type of `update` as follows:

```
update  : (((ref (◯ ⊤)) at server)  ⊃  ◯ ⊤  ⊃  ◯ ⊤) < client >
```

The type `((ref (◯ ⊤)) at server) ⊃ ◯ ⊤ ⊃ ◯ ⊤` says that `update` takes a reference to a computation, located at the server, along with a computation, and produces a computation. The Agda function `A < w >` takes a `Type` and a `World` and produces a `Set`; here, it is used to say that the whole type is interpreted relative to the client.

The above polymorphic `update'` is typed as follows:

```
update' : (□(∀₅ (λ w2 → ref (◯ ⊤) at w2  ⊃  ◯ ⊤ ⊃ ◯ ⊤))) <*>
```

where the postfix symbol `<*> :  Type → Set` means that the proposition is true in all worlds (i.e. `A <*>` means the Agda type `{w :  World} → A < w >`). The outer $\Box$ binds the "client" world (i.e. the world for the computations in $\bigcirc\ \top \supset \bigcirc\ \top$); the inner $\forall_5$ binds the "server" world (i.e. the world of the `ref (◯ ⊤)`.

3.2 Interpretation

We define an interpretation function `A < w >` interpreting a type `A` and a world `w` as an Agda classifier (a `Set`). Then the proof of an HL5 judgement `A1 < w1 > ...  ⊢ C < w >` is an Agda function of type `A1 < w1 > ...  → C < w >`. This interpretation is a *constructive Kripke semantics*—i.e. a Kripke semantics of an intuitionistic modal logic in intuitionistic first-order logic, relative to the Kripke structure given by the type `World`.[1] The interpretation is defined as as follows:

[1]Technically, we omit two of the pieces of a Kripke structure: `World` is the set of states, but because we are representing S5, we can elide the accessibility relation, and because we do not have uninterpreted base `Types`, we do not need an interpretation of them.

```
_<_> : Type → World → Set
(ref A) < w > = Ref w (A < w >)
(◯ A) < w > = IO w (A < w >)
(A at w') < _ > = A < w' >
(A ⊃ B) < w > = A < w > → B < w >
(A ∨ B) < w > = Either (A < w >) (B < w >)
(∀₅ A) < w >   = (w' : World)  →  (A w') < w >
(∃₅ A) < w >   = Σ \ (w' : World)  →  (A w') < w >
```

The main action of the translations is to annotate `ref` and ◯ with the world at which the type is being interpreted. The hybrid connective `at` interprets its body at the specified world, ignoring the current world. Otherwise, the translation interprets each connective as the corresponding Agda `Set`-former, with the translation applied recursively. Note that the interpretation of ⊃ is simply →: when giving a Kripke semantics for intuitionistic logic in a *classical* meta-language, it is necessary to interpret `A ⊃ B` as if it were boxed, by quantifying over future worlds; but because our meta-language is intuitionistic, this is not necessary here.

The function `<*> : Type → Set` is defined to be the Agda set `{w : World} → A < w >`—`A` is true in all worlds, with the world itself an implicit argument to the function. Agda verifies that the modal types of `update` and `update'` reduce to the explicit types given above.

This formalization has several benefits: we can immediately use Agda to program in the modal logic, and existing Agda code can be used at modal types. For example, one can check that this definition validates all of the rules of intuitionistic S5 [28], by implementing the proof that the Simpson rules are sound for the Kripke semantics. The untethered elimination rule for disjunction and a projective elimination rule for `at` are defined as follows:

```
casev : ∀ {A B C w w'} →  A ∨ B < w >
      → (A < w > → C < w' >)  →  (B < w > → C < w' >)
      → C < w' >
casev (Inl e) b1 b2 = b1 e
casev (Inr e) b1 b2 = b2 e

unatv : ∀ {A w w'}  →  (A at w') < w >  →  A < w' >
unatv x = x
```

Additionally, we can see that the return and bind operations on the monad have their expected types at any world:

```
hl5ret : ∀ {A} →  A ⊃ ◯ A  <*>
hl5ret = return

hl5bind : ∀ {A B} →  ◯ A ⊃ (A ⊃ ◯ B) ⊃ ◯ B  <*>
hl5bind =  _>>=_
```

The type of bind insists that all three ◯'s be at the same world: sequencing tethers the premise to the conclusion.

Having considered return and bind, it is natural to ask what hybrid type can we ascribe to the `get` operation on the indexed monad `IO w A`. If we try defining

```
hl5get : ∀ {A w1 w2} → ((◯ A) at w1) ⊃ ((◯ A) at w2) <*>
```

we see that `hl5get` must transform `IO w1 (A < w1 >)` into `IO w2 (A < w2 >)`. L5 `get` satisfies this requirement if `A` is a *constant function* of its world argument, in which case `A < w1 >` is the same Agda `Set` as `A < w2 >`.

We characterize constant modal types by the property that they yield the same `Set` for any two arguments:

```
Constant : Type → Set1
Constant A = ∀ {w w'} → EqSet (A < w >) (A < w' >)
```

Here `EqSet` is an Agda relation expressing that the two `Sets` are equal classifiers (in fact, we need a notion of equality that compares the bodies of Π and Σ on all arguments, which we borrow from OTT [4]); it is equipped with an operation `coerce : ∀ {A B} → EqSet A B → A → B`. It is simple to prove that `A at w` is constant, and that the connectives $\vee \supset \forall_5\ \exists_5$ preserve constantness. Neither `ref` nor $\bigcirc$ is constant, as their interpretation directly mentions the world. Thus, the constant types are those where all `ref`s and $\bigcirc$'s are guarded by an `at`.

Now, we can ascribe `get` the following monadic type:

```
hl5get : ∀ {A w1 w2} → Constant A → ((◯ A) at w1) ⊃ ((◯ A) at w2) <*>
hl5get con e = get e »= \ v → return (coerce con v)
```

`hl5get` is equivalent to `get`, using the monad laws and the fact that coercion based on a proof of `EqSet` is the identity (at least up to extensional equality),

4 ML5

In this section, we give inductive definitions of the syntactic apparatus of ML5: First, we define the syntax of types. Next, we represent programs using an *intrinsic encoding*, which represents only well-typed syntax (i.e. typing derivations or natural deduction proofs). Variables are represented as well-scoped de Bruijn indices—pointers into a typing context, which is an explicit parameter to the typing judgements. The static semantics requires an auxiliary definition of a *mobility* judgement on types, which is defined below.

4.1 Types

```
data Type5 : Set where
  _⇀_ _∨_  : Type5 → Type5 → Type5
  ∀₅ ∃₅ : (World → Type5) → Type5
  _at_ : Type5 → World → Type5
  ref  : Type5 → Type5
  ☘   : (World → Type5) → Type5
```

The type $\rightharpoonup$ represents partial functions, and the type $\vee$ represents sums. The types $\forall_5\ \exists_5$ cdat and `ref` are the same as in HL5. There is an additional type constructor "shamrock", rendered here as ☘, which is a $\Box$-like modality, but its rules in the ML5 proof theory are subtly different than the rules for $\forall_5$ `(λ w → A at w)`.

We represent types with free world variables as Agda functions from worlds to types. This is permissible because `World` is defined prior to type (i.e. we are using Weak HOAS[11, 16]). If `World` is chosen to be a base type in Agda, then it adequately represents ML5 types as in Murphy [23]. If instead `World` is chosen to be an inductive type, this representation yields a language with type-level case analysis over worlds—i.e. one could define types whose structure varies depending on their world, such as $\exists_5$ `(λ w → if w = server then nat else bool)`; we leave an exploration of the practical uses of this alternative to future work.

4.2 Mobility

ML5's notion of mobility identifies constant functions, analogously to the `Constant` relation on HL5 types above.

```
data Mobile5 : Type5 → Set where
  mat5 : ∀ {A w} → Mobile5 (A at w)
  m𝔼5 : ∀ {A} → Mobile5 (𝔼 A)
  -- no rule for ⇀ or refs
  m∨5  : ∀ {A B} → Mobile5 A → Mobile5 B → Mobile5 (A ∨ B)
  m∀5  : ∀ {A} → ((w' : _) → Mobile5 (A w')) → Mobile5 (∀₅ A)
  m∃5  : ∀ {A} → ((w' : _) → Mobile5 (A w')) → Mobile5 (∃₅ A)
```

To foreshadow the interpretation: 𝔼 is always mobile, essentially because $\forall_5$ (λ w → (A w) at w) is constant. Functions are never mobile, because they hide a ◯ in their domain, and ◯ is not constant.

4.3 Typing judgements

ML5 judgements have the form Γ ⊢ `e : A [ w ]` and Γ ⊢ `v :: A [ w ]`. These judgements mean that the expression `e` and the value `v` are well-formed with modal type `A` at world `w`. Here Γ contains assumptions `x:A[w]` and `u ∼ w.A`. The former, a *value hypothesis*, means that `x` stands for a value of type `A` at world `w`. The latter, a *valid hypothesis*, means that `u` stands for a value of type `A` that makes sense at all worlds (`w` is bound in `A`). In the operational semantics, we will substitute a proof of the judgement `w:world ⊢ v :: (A w)[ w ]` for a valid variable.

We combine both of the above judgements into one Agda type, defining a relation Γ ⊢ γ. Here Γ is a list of hypotheses, which are either `value (A [ w ])` or `valid (w.A)` and γ is a conclusion, which is either `exp (A [ w ])` (for expressions) or `value (A [ w ])` (for values). As in the syntax of types, `w.A` is represented by an Agda function from worlds to types. ⊢ binds more tightly than →, so we can write e.g. Γ ⊢ `C1` → Γ ⊢ `C2` for `(`Γ ⊢ `C1) → (`Γ ⊢ `C2)`.

We define the notation `A [ w ]` to mean the pair of `A` and `w`, and we define sum types for hypotheses and conclusions as follows:

```
data Hyp : Set where
  value : (Type5 × World) → Hyp
  valid : (World → Type5) → Hyp

data Conc : Set where
  exp  : (Type5 × World) → Conc
  value : (Type5 × World) → Conc

Ctx = List Hyp
```

Term variables `x` and `u` are represented by well-scoped de Bruijn indices—pointers into Γ:

```
data _∈_ {A : Set} : A → List A → Set where
 i0 : {α : A}   {Γ : List A} → α ∈ (α :: Γ)
 iS : {α α' : A} {Γ : List A} → α ∈ Γ → α ∈ (α' :: Γ)
```

The typing rules are defined in Figure 1. The first rule says that `(▷ x)` is a value if `x` is a de Bruijn index for a value assumption in Γ. The next rule represents the application of a valid assumption in Γ to a world `w`; we use propositional equality `Id` to state that the conclusion type is `A w` because otherwise

the higher-order conclusion blocks pattern-matching. `lam` takes an expression in an extended context to a value of function type; this rule expresses the idea that a function of type `A ⇀ B` in a world `w` is an expression of type `B` at `w`, hypothetically in a variable standing for a value of type `A` at `w`. Sums are introduced by commuting the world with the connective. `hold` switches worlds to introduce an `at`; `wlam` and `wpair` introduce quantifiers in the usual way; the world in the conclusion does not change (also, note the parentheses, which in Agda's notation for datatype constructors are the only difference between the two rules: the premise of `wlam` is a function, whereas `wpair` has two premises). However, note that the body of a `wlam` is a value, not an expression—ML5 has a value restriction on world quantification, to support type inference in the style of ML. `sham` both quantifies over a new world and switches the world of the conclusion—indeed, this is the derived intro rule that one would expect for `⦂ A = ∀₅ (λ w → (A w) at w)`.

`val` injects values into expressions, whereas `let` sequences the evaluation of two expressions. `put` runs an expression of mobile type and then binds the resulting value as a valid assumption. `app` and `wapp` eliminate functions and universals, with the world along for the ride. The remaining rules are pattern-matching-style elimination rules. In these rules, the world in the conclusion `C [ w ]` is *tethered* to the world in the principal premise, as discussed in the introduction.

This system is the ML5 internal language described in Section 4.3 of [23] with one minor change: we have eliminated the syntactic class of valid values, which allowed validity to appear as a conclusion as well as as a hypothesis—valid values are unnecessary because validity is invertible on the right. An alternate formalization with valid values is included in the companion code.

5 Semantics

Our semantics explains the meaning of an ML5 program by translation into HL5. As discussed above, our translation clarifies the essential difference between the role that the world plays in the judgements `value (A [ w ])` and `exp (A [ w ])`: a value hypothesis or conclusion `value (A [ w ])` is interpreted as a *pure term* of HL5 type `(eff A) < w >`, where `eff A` is a monadic translation of `A`. Thus, the role of the world `w` is only to describe where the resources in `A` may be used and where the computations must be run. On the other hand, an expression `exp (A [ w ])` is interpreted as a *monadic computation* of type `(◯ (eff A)) < w >`, so the world determines both the site at which the effectful computation must be run and where the resources/computations in `A` may be used.

5.1 Type Translation

The type translation from ML5 types to HL5 translates `⦂` to HL5 `□`, adds a `◯` to the codomain of `⇀` (as in any monadic translation [21]), and is defined compositionally otherwise. The ML5 connective `∀₅` has a value-restriction, so there is no `◯` inserted in its body. Note that Agda allows datatype constructors to be overloaded, so we can have both ML5 types and HL5 types in scope at once.

```
eff : Type5 → Type
eff (A ⇀ B) = eff A ⊃ ◯(eff B)
eff (A ∨ B) = eff A ∨ eff B
eff (∀₅ A) = ∀₅ (λ w → (eff (A w)))
eff (∃₅ A) = ∃₅ (λ w → eff (A w))
eff (A at w) = (eff A) at w
eff (ref A) = ref (eff A)
eff (⦂ A) = ∀₅ (λ w → ((eff (A w)) at w))
```

```
data _⊢_ (Γ : Ctx) : Conc → Set where

  -- values
  ▷ : ∀ {A w}
   → value (A [ w ]) ∈ Γ
   → Γ ⊢ value (A [ w ])

  ▷v : ∀ {w A C}
   → valid A ∈ Γ  →  Id C (A w)
   → Γ ⊢ value (C [ w ])

  lam : ∀ {A B w}
   → (Γ ,, value (A [ w ])) ⊢ exp (B [ w ])
   → Γ ⊢ value (A ⇀ B [ w ])

  inl : ∀ {A B w}
   → Γ ⊢ value (A [ w ])
   → Γ ⊢ value (A ∨ B [ w ])

  inr : ∀ {A B w}
   → Γ ⊢ value (B [ w ])
   → Γ ⊢ value (A ∨ B [ w ])

  hold : ∀ {A w w'}
   → Γ ⊢ value (A [ w' ])
   → Γ ⊢ value ((A at w') [ w ])

  wlam : ∀ {A w}
   → ( (w' : _) → Γ ⊢ value ((A w') [ w ]) )
   → Γ ⊢ value (∀₅ A [ w ])

  wpair : ∀ {A w}
   → (w' : World)
   → Γ ⊢ value ((A w') [ w ])
   → Γ ⊢ value ((∃₅ A) [ w ])

  sham : ∀ {A w'}
   → ((w : _) → Γ ⊢ value (A w [ w ]))
   → Γ ⊢ value (☒ A [ w' ])

  --- expressions

  val : ∀ {L}
   → Γ ⊢ value L
   → Γ ⊢ exp L

  lete : ∀ {A C w}
   → Γ ⊢ exp (A [ w ])
   → (Γ ,, value (A [ w ])) ⊢ exp (C [ w ])
   → Γ ⊢ exp (C [ w ])

  get5 : ∀ {A w w'}
    → Γ ⊢ exp (A [ w' ])  →  Mobile5 A
    → Γ ⊢ exp (A [ w ])

  put : ∀ {A C w}
    → Γ ⊢ exp (A [ w ])  →  Mobile5 A
    → (Γ ,, valid (\ _ → A)) ⊢ exp (C [ w ])
    → Γ ⊢ exp (C [ w ])

  app : ∀ {A B w}
    → Γ ⊢ exp (A ⇀ B [ w ])  →  Γ ⊢ exp (A [ w ])
    → Γ ⊢ exp (B [ w ])

  wapp : ∀ {A w}
    → Γ ⊢ exp (∀₅ A [ w ])  →  (w' : World)
    → Γ ⊢ exp ((A w') [ w ])

  case : ∀ {A B C w}
    → Γ ⊢ exp (A ∨ B [ w ])
    → (Γ ,, value (A [ w ]) ⊢ exp (C [ w ]))
    → (Γ ,, value (B [ w ])  ⊢ exp (C [ w ]))
    → Γ ⊢ exp (C [ w ])

  wunpack : ∀ {A w C}
    → Γ ⊢ exp ((∃₅ A) [ w ])
    → ((w' : _) →
       Γ ,, value ((A w') [ w ]) ⊢ exp (C [ w ]))
    → Γ ⊢ exp (C [ w ])

  leta : ∀ {A w w' C}
    → Γ ⊢ exp ((A at w') [ w ])
    → (Γ ,, value (A [ w' ]) ⊢ exp (C [ w ]))
    → Γ ⊢ exp (C [ w ])

  lets : ∀ {A w C}
    → Γ ⊢ exp (☒ A [ w ])
    → (Γ ,, valid A ⊢ exp (C [ w ]))
    → Γ ⊢ exp (C [ w ])
```

Figure 1: ML5 Static Semantics

A simple induction verifies that mobile ML5 types translate to constant HL5 types:

```
eff-mobile : ∀ {A} → Mobile5 A → Constant (eff A)
```

5.2 Term translation

Next, we interpret hypotheses and conclusions. Values and expressions are interpreted as described above (note that for `L = A [ w ]`, `fst L = A` and `snd L = w`). A valid hypothesis is interpreted as the Agda set `(w : World) → (interp-hyp (value (A w [ w ])))`, though to appease the termination checker we have to unroll the definiton of `interp-hyp`.

```
interp-hyp : Hyp → Set
interp-hyp (value L) = (eff (fst L)) < (snd L) >
interp-hyp (valid A) = (w : World) → (eff (A w)) < w >

interp-conc : Conc → Set
interp-conc (value L) = interp-hyp (value L)
interp-conc (exp L) = (◯ (eff (fst L))) < (snd L) >
```

Next, we interpret the syntax (Figure 2). The Agda type `Everywhere P xs` represents a list with one element of type `P x` for each element `x` in `xs`; we use this to represent a substitution of semantic values for syntactic variables. The usual propositional connectives are interpreted in the standard way, by sequencing evaluation if necessary and then applying the corresponding Agda introduction or elimination form. Subexpressions that are under bound variables are interpreted in an extended context (e.g. `e` in `lam e`). `List.EW.there` looks up a de Bruijn index into Γ in a substitution of type `Everywhere P` Γ.

`sham` is interpreted as an Agda function, which is applied in the translation of the `▷v` rule for using valid variables. There are no Agda term constructors for `at`, so the translation simply proceeds inductively. The quantifiers are interpreted by supplying the world arguments in the syntax, rather than by extending the substitution (recall that the proof terms `wlam` and `wunpack` contain Agda functions). Recall that the body of `wlam` is already a value, so no further evaluation is necessary.

`val` and `lete` are interpreted directly as return and bind. `get5` is translated as a `get` in the target, followed by a coercion by the mobility proof. `put e` extends the substitution with the value of `e`, applying the mobility proof to the value so it can be used at any other world.

Agda verifies that the interpretation is type correct and total, establishing that the interpretation is total and type-preserving.

6 Revised ML5

In this section, we simplify and generalize the ML5 source language based on the above semantics. First, our analysis has shown that `Mobile5 A` really means that `A [ w ]` and `A [ w' ]` are equal types for any `w` and `w'`. We can internalize this principle as a value coercion `vshift` in Figure 3. This makes it possible to implement some additional programs without communication. For example, consider a function

```
move : ∀ {A w w'} →  Mobile5 A  →  (value (A [ w ]) :: []) ⊢ exp (A [ w' ])
move m =  (get5 (val (▷ i0)) m)
```

Operationally, this is quite inefficient: it sends the value of the variable from `w'` to `w`, as the environment of the `val (▷ i0)` closure, and `w` then returns this value back to `w'`. This whole process is unnecessary,

```
eval : ∀ {Γ L} → Γ ⊢ L → Everywhere interp-hyp Γ → interp-conc L
-- usual connectives
eval (▷ x) σ = (List.EW.there σ x)
eval (lam e) σ = λ x → eval e (x E:: σ)
eval (app e1 e2) σ =  (eval e1 σ) >>= λ f → (eval e2 σ) >>= λ a → f a
eval (inl v) σ = Inl (eval v σ)
eval (inr v) σ = Inr (eval v σ)
eval (case e e1 e2) σ = eval e σ >>= docase where
  docase : Either _ _ → _
  docase (Inl x) = eval e1 (x E:: σ)
  docase (Inr y) = eval e2 (y E:: σ)
-- ⟀
eval (▷v{w} x Refl) σ =  (List.EW.there σ x) w
eval (sham v) σ =  λ w' → (eval (v w') σ)
eval (lets e e') σ = eval e σ >>= λ x → eval e' (x E:: σ)
-- at
eval (hold v) σ = eval v σ
eval (leta e e') σ = eval e σ >>= λ x → eval e' (x E:: σ)
-- ∀ and ∃
eval (wlam v) σ = λ w → eval (v w) σ
eval (wapp e w) σ = eval e σ >>= λ f → return (f w)
eval (wpair w v) σ = w , eval v σ
eval (wunpack e e') σ = eval e σ >>= λ p → eval (e' (fst p)) (snd p E:: σ)
-- monad operations and world movement
eval (val v) σ = return (eval v σ)
eval (lete e1 e2) σ = eval e1 σ >>= λ x → eval e2 (x E:: σ)
eval (get5{A} e mob) σ =  get (eval e σ) >>= λ v → return (coerce (eff-mobile mob) v)
eval (put e mob e') σ = eval e σ >>= λ v → eval e' ((λ w' -> coerce (eff-mobile mob) v) E:: σ)
```

Figure 2: Interpretation of the syntax

as the value was available at `w'` to begin with! However, it does not seem possible to implement this function without communication in ML5. In our revised ML5, as in HL5, it can be implemented as a simple type coercion:

```
move : ∀ {A w w'} →  Mobile5 A  →  (value (A [ w ]) :: []) ⊢ exp (A [ w' ])
move m = val (vshift (▷ i0) m)
```

Second, in HL5, the worlding of values always "gets out of the way" because it commutes with type constructors down to `ref` and ◯. In Figure 3, we add these non-local elimination rules for values of each connective to the source. For example, we add the untethered case rule for values of sum type and a projective elimination rule for `at` and $\forall_5$ values.

We could additionally add elimination rules like `case`, `split`, etc. to the syntactic class of "values"—i.e. we could admit that we are really dealing with a syntactic class of pure terms:

```
casev/val : ∀ {A B C w' w} → Γ ⊢ value (A ∨ B [ w ])
  → (Γ ,, value (A [ w ]) ⊢ val (C [ w' ])) → (Γ ,, value (B [ w ]) ⊢ val (C [ w' ]))
  → Γ ⊢ val (C [ w' ])
```

However, in an operational semantics where worlds and types are erased at run-time, `wappv` and `unatv` will not create any real redexes at run-time, whereas `casev/val` will. Thus a reasonable design choice for ML5 would be to allow `wappv` and `unatv` but not the corresponding `case` rule.

New value rules:

```
vshift : ∀ {A w w'} → Γ ⊢ value (A [ w ]) →  Mobile5 A → Γ ⊢ value (A [ w' ])
wappv : ∀ {A w} → Γ ⊢ value (∀₅ A [ w ])  →  (w' : World) → Γ ⊢ value ((A w') [ w ])
unatv : ∀ {A w w'} → Γ ⊢ value ((A at w') [ w ]) → Γ ⊢ value (A [ w' ])
```

New expression rules:

```
casev : ∀ {A B L w} → Γ ⊢ value (A ∨ B [ w ])
  → (Γ ,, value (A [ w ]) ⊢ exp L) → (Γ ,, value (B [ w ])  ⊢ exp L)
  → Γ ⊢ exp L
wunpackv : ∀ {A w L} → Γ ⊢ value ((∃₅ A) [ w ])
         → ((w' : _) → Γ ,, value ((A w') [ w ]) ⊢ exp L) → Γ ⊢ exp L
```

Remove rules put, ▷v, lets, sham, wapp, leta, case, wunpack.

Figure 3: Revised ML5

It is simple to extend the semantics to these new constructs, as they were derived from it:

```
eval (vshift e m) σ = coerce (eff-mobile m) (eval e σ)
eval (wappv e w') σ = eval e σ w'
eval (unatv e ) σ = eval e σ
eval (casev v e1 e2) σ = docase (eval v σ) where
  docase : Either _ _ → _
  docase (Inl x) = eval e1 (x E:: σ)
  docase (Inr y) = eval e2 (y E:: σ)
eval (wunpackv v e') σ = let p = eval v σ in eval (e' (fst p)) (snd p E:: σ)
```

Derived Forms Once we have added the aforementioned rules, we can remove rules put, ▷v, lets, sham, wapp, leta, case, wunpack, which become derivable. Shamrock is defined using $\forall_5$ and at in the straightforward way: ♣ A = $\forall_5$ (λ w → A w at w). Validity is defined as a value assumption of shamrock type: valid A = value ((♣ A) [dummy]), where dummy is any arbitrary world—♣ A is mobile, and therefore constant, but because all value assumptions are worlded, we must pick one.

The term ▷v, which represents a use of a valid hypothesis, is derived by eliminating a shamrock. The term put is derived by introducing a shamrock, using vshift to retype the assumption in the body—we use letv to refer to the substitution principle plugging a value in for a variable, and we use weaken for weakening in the de Bruijn syntax.

```
▷v : ∀ {Γ A C w} → valid A ∈ Γ  →  Id C (A w) → Γ ⊢ value (C [ w ])
▷v i Refl = unatv (wappv (▷ i) _)

put : ∀ {Γ A C w} → Γ ⊢ exp (A [ w ])  →  Mobile5 A
    → (Γ ,, valid (\ _ → A)) ⊢ exp (C [ w ]) → Γ ⊢ exp (C [ w ])
put e m e' = lete e (letv (wlam (\ w' → hold (vshift (▷ i0) m))) (weaken (extend⊆ iS) e'))
```

The remaining derivabilities show that the old rules for the connectives are derivable using the generalized ones.

7 Related Work

Murphy [23] describes ML5 and related languages, such as work by Jia and Walker [19].

Altenkirch and McBride [3], Benke et al. [9], Chlipala [12], Crary [14] describe other uses of universes and semantic embeddings in type theory, though they do not consider embedding a modal type system. We have used the same technique for embedding a hybrid type system in Agda in previous work [20]. Our technique is quite similar to that of Allen [2], who defines modal types as display forms for NuPRL types. The technical difference is that Allen considers the modal types simply as notation, whereas in our approach the modal types are data, equipped with a translation to meta-language types. This shows how to achieve similar convenience of notation, without requiring separate display-form facilities. Avron et al. [8] consider representations of modal logics in LF [18], some of which use world-indexed judgements to track scoping. We also use a world-indexed type family `A < w >`, but this relation is defined semantically (by interpretation into Agda) rather than syntactically (by inference rules).

At the core, our interpretation reduces ML5 to L5, a language with an indexed monad `IO w A` of computations at a place. Indexed monads have been studied in a variety of previous work, including Abadi et al. [1], Atkey [6], Nanevski et al. [24], Russo et al. [27]. However, to interpret ML5, we require the programming language to provide quantification over the indices to the monad, which DCC [1], for example, does not provide. It would be interesting to adapt our work to these other settings, using a modal logic to manage the indices to the monad.

8 Conclusion

While we have used Agda for our development, we conjecture that the work described in this paper could be carried out with similar effort in Coq [13], as we have not used very complicated dependent pattern matching. However, in future work we would like to embed proof-based access control following PCML5 [7], which will require a modal universe with dependent types. Dependently typed universes are easiest to represent using induction-recursion [17], which Agda supports but Coq does not.

In future work, we also plan to complete a proof that the operational semantics of ML5 are sound for the denotational semantics. We have formalized the operational semantics of λ_5 and an operational semantics for computations. We have also proved soundness, assuming a standard compositionality lemma (substitution of interpretations is the interpretation of the substitution), which we are in the process of formalizing. β-reduction in Agda validates the β-steps for functions, sums, etc. in the source. Because compositionality is really a property only of the binding structure of the language and the semantics, not of the particular language constructs, it should be possible to implement compositionality in a datatype-generic manner, as in Chlipala [12]. We also leave the question of full abstraction to future work.

Acknowledgements We thank Jason Reed and Rob Simmons for discussions about this article, and the anonymous reviewers for their helpful comments.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL*, pages 147–160. ACM Press, 1999.

[2] S. F. Allen. From dy/dx to []P: A matter of notation. In *Proceedings of the Conference on User Interfaces for Theorem Provers*, 1998.

[3] T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In *IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl*, 2003.

[4] T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now! In *Programming Languages meets Program Verification Workshop*, 2007.

[5] C. Areces, P. Blackburn, and M. Marx. Hybrid logics: Characterization, interpolation and complexity. *Journal of Symbolic Logic*, 66(3):977–1010, 2001.

[6] R. Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19(3–4):335–376, 2009.

[7] K. Avijit, A. Datta, and R. Harper. Distributed programming with distributed authorization. In *ACM SIGPLAN-SIGACT Symposium on Types in Language Design and Implementation*, 2010.

[8] A. Avron, F. Honsell, M. Miculan, and C. Paravano. Encoding modal logics in logical frameworks. *Studia Logica*, 60(1):161–208, 1998.

[9] M. Benke, P. Dybjer, and P. Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10(4):265–289, 2003.

[10] P. Benton, G. M. Bierman, and V. C. V. D. Paiva. Computational types from a logical perspective. *Journal of Functional Programming*, 8(2):177–193, 1998.

[11] A. Bucalo, M. Hofmann, F. Honsell, M. Miculan, and I. Scagnetto. Consistency of the theory of contexts. *Journal of Functional Programming*, 16(3):327–395, May 2006.

[12] A. Chlipala. A certified type-preserving compiler from λ-calculus to assembly language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[13] Coq Development Team. *The Coq Proof Assistant Reference Manual*. INRIA, 2007. Available from `http://coq.inria.fr/`.

[14] K. Crary. *Type Theoretic Methodology for Practical Programming Languages*. PhD thesis, Cornell University, 1998.

[15] H. Curry. The elimination theorem when modality is present. *Journal of Symbolic Logic*, 17:249–265, 1952.

[16] J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *International Conference on Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 124–138, Edinburgh, Scotland, 1995. Springer-Verlag.

[17] P. Dybjer and A. Setzer. Indexed induction-recursion. In *Proof Theory in Computer Science*, pages 93–113. Springer, 2001.

[18] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1), 1993.

[19] L. Jia and D. Walker. Modal proofs as distributed programs. In *European Symposium on Programming*, volume 2986 of *Lecture Notes in Computer Science*. Springer, April 2004.

[20] D. R. Licata and R. Harper. A universe of binding and computation. In *ACM SIGPLAN International Conference on Functional Programming*, 2009.

[21] E. Moggi. Notions of computation and monads. *Information And Computation*, 93(1), 1991.

[22] J. Morgenstern and D. R. Licata. Security-typed programming within dependently-typed programming. Available from `http://www.cs.cmu.edu/~drl`, April 2010.

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

[24] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare Type Theory. In *ACM SIGPLAN International Conference on Functional Programming*, pages 62–73, Portland, Oregon, 2006.

[25] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[26] F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In H. Ganzinger, editor, *International Conference on Automated Deduction*, pages 202–206, 1999.

[27] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in haskell. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 13–24, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-064-7. doi: http://doi.acm.org/10.1145/1411286.1411289.

[28] A. Simpson. *The Proof Theory and Semantics of Intuitionistic Modal Logic*. PhD thesis, University of Edinburgh, 1993.