Thinking Outside the □: Verified Compilation of ML5 to JavaScript

Joomy Korkut
Advisor: Daniel R. Licata
Wesleyan University

Abstract

Curry-Howard correspondence describes a language that corresponds to propositional logic. Since modal logic is an extension of propositional logic, then what language corresponds to modal logic? If there is one, then what is it good for? Murphy's dissertation[1] argues that a programming language designed based on modal type systems can provide elegant abstractions to organize local resources on different computers. In this thesis, I limit his argument to simple web programming and claim that a modal logic based language provides a way to write readable code and correct web applications. To do this, I defined a minimal language called ML5 in the Agda proof assistant and then implemented a compiler to JavaScript for it and proved its static correctness. The compiler is a series of type directed translations through fully formalized languages, the last one of which is a very limited subset of JavaScript. As opposed to Murphy's compiler, this one compiles to JavaScript both on the front end and back end through Node.js. Currently the last step of conversion to JavaScript is not entirely complete. We have not specified the conversion rules for the modal types, and network communication only has a partially working proof-of-concept.

Background

Modal logic is a broad field that includes various kinds of logic that deal with relational structures that have different perspectives of truth. We call these perspectives of truth, "possible worlds". Each world holds a possibly different set of truths. Now we do not have the "A true" judgment, we specify the world and say "A true at world w". Our notation for that is $A{<}w{>}$.

The intuitionistic modal logic IS5$^{\cup}$ allows data transition from any world to another. Traditionally modal logic has the $\Box$ connective, which means a proposition is correct for all world, and the $\Diamond$ connective, which means a proposition is correct for some world. We replace them with the hybrid connectives at, $\forall$ and $\exists$, such that $\Box P = \forall\omega.P\ \mathsf{at}\ \omega$ and $\Diamond P = \exists\omega.P\ \mathsf{at}\ \omega$.

We then define the language Lambda 5 based on the proof terms of IS5$^{\cup}$, and we include a notion of mobility that oversees what can be transferred between worlds. The relationship between modal logic rules and proof terms in Lambda 5 should resemble how propositional logic and simply typed lambda calculus are related in Curry-Howard correspondence, i.e. modal propositions will be types in Lambda 5, and proof trees will be Lambda 5 expressions.

Type-Directed Translation

Our compiler has 5 conversion steps before JavaScript:

(1) **ML5**: an Agda formalization of Lambda 5

(2) **Continuation-passing style:** Considering that most actions in JavaScript are run through callbacks, this process is necessary to move us closer to JavaScript, our final target language.

(3) **Closure conversion**: We eventually want to hoist all lambdas in a program to the top, so that we can call them by their names during network communication. However, this is not possible because these functions contain bound variables from previous definitions. That is why we create closures to get rid of these bound variables.

(4) **Lambda lifting**: Now that functions do not have any other bound variables other than the argument of the function they are in, we can hoist the functions.

(5) **Monomorphic**: Before conversion to JavaScript, we have to monomorphize valid values into values in specific worlds.

Formalization of JavaScript

To prevent runtime errors in the code we generate, we will formalize a subset of JavaScript that enforces certain type and context restrictions. We are defining three syntactic categories for our formalization: statements, function statements and expressions.

$\mathsf{Stm}\ \Gamma < \mathsf{w} >$ should read "the statement under the context Γ in the world w".

$\mathsf{FnStm}\ \Gamma \Downarrow \gamma \wr \mathsf{m}\sigma < \mathsf{w} >$ should read "the function statement under the context Γ that extends the context with γ and has returned the function with type $\mathsf{m}\sigma$, in the world w". FnStm can only be used in lambda terms.

$\Gamma \vdash \tau < \mathsf{w} >$ should read "the expression under the context Γ, of the type τ, in the world w".

Conversion to JavaScript

We are defining functions to convert continuation expressions and expressions to JavaScript expressions and function statements.

```
convertCont : ∀ {Γ Δ Φ}
  → {s : only client (convertCtx Γ) ⊆ Δ}
  → {s' : only server (convertCtx Γ) ⊆ Φ}
  → (w : World)
  → Γ ⊢ᵐ ⋆< w >
  → Σ _ (λ δ → FnStm Δ ⇓ δ ≀ nothing < client >)
    × Σ _ (λ φ → FnStm Φ ⇓ φ ≀ nothing < server >)
convertValue : ∀ {Γ Δ Φ τ w}
  → {s : only client (convertCtx Γ) ⊆ Δ}
  → {s' : only server (convertCtx Γ) ⊆ Φ}
  → Γ ⊢ᵐ ↓ τ < w >
  → (only w (convertCtx Γ)) ⊢ⱼ (convertType {w} τ) < w >
    × Σ _ (λ δ → FnStm Δ ⇓ δ ≀ nothing < client >)
    × Σ _ (λ φ → FnStm Φ ⇓ φ ≀ nothing < server >)
```

After we get a FnStm, we will put it in an anonymous function and call that immediately, which keeps everything we defined in the local context and encapsulates our program. At the end, we can have a huge function composition that serves as a compiler pipeline from ML5 to JavaScript. Its definition is as follows:

```
compileToJS : [] ⊢₅ 'Unit < client > → String × String
compileToJS = (clientWrapper *** serverWrapper)
  ∘ (stmSource *** stmSource)
  ∘ LiftedMonomorphicToJS.entryPoint
  ∘ LiftedMonomorphize.entryPoint
  ∘ LambdaLifting.entryPoint
  ∘ CPStoClosure.convertCont
  ∘ ML5toCPS.convertExpr (λ v → CPS.Terms.'halt)
```

References

[1] Tom Murphy VII. *Modal types for mobile code.* PhD thesis, Carnegie Mellon University, 2008.