Wesleyan University The Honors College

Thinking Outside the □:
Verified Compilation of ML5 to JavaScript

by

Joomy Korkut

A thesis submitted to the
faculty of Wesleyan University
in partial fulfillment of the requirements for the
Degree of Bachelor of Arts
with Departmental Honors in Mathematics and Computer Science

Middletown, Connecticut April 2017

"Forgotten were the elementary rules of logic, that extraordinary claims require extraordinary evidence and that what can be asserted without evidence can also be dismissed without evidence."

Christopher Hitchens

Acknowledgements

First and foremost, I would like to thank Dan Licata, my research advisor. He is the primary reason that I had an opportunity to get into computer science research, and I am immensely grateful for his patience and humility throughout the years that we worked together. Every undeserved "Good work!" I got from him was a great source of motivation for me.

I would like to express my gratitude to Norman Danner and Jeff Epstein for reading and evaluating my thesis. Additionally, the great computer science and mathematics courses I have taken with James Lipton, Norman Danner, Jeff Epstein, Cameron Donnay Hill and David Pollack have sparked my interest in research, so I would like to thank them for that.

I would like to thank my dear colleague and friend Maksim Trifunovski for his support and comradery throughout the years we did research and worked as course assistants together, not to mention his endless supply of *rakija*.

I am thankful to Pi, Emily, Molly, Kivanc, Damlasu, Isin Ekin, and my family for the emotional support they provided by putting up with me babbling ceaselessly about linguistics and politics. Finally, I would like to thank Cloie for helping me find a logic pun for my thesis title.

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[18] 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. (targeting 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.

Contents

1. Introduction

The field of web development was drastically different a decade ago from what it is today. AJAX and frameworks like jQuery and Prototype had just been introduced to the world and programmers had just started to use JavaScript for things other than tacky effects. Concepts like single-page application and server-side JavaScript were not around. As these things changed and JavaScript became a sine qua non in web programming, programmers started to complain more and more about the idiosyncracies of this language. Despite the inconvenience of programming with JavaScript, people still had to use it because there were no serious alternatives. This issue came to be known as the JavaScript problem.[12] The common solution to it was to create higher level languages or different syntaxes that compiled back to JavaScript.

This thesis stems from the same necessity of programming in a language that is more sensible than JavaScript. Another problem we hope to solve is that current web applications require too much boilerplate code to do network communication with the server. This makes writing single-page applications an annoying task. The language we will define in this thesis will not attempt solving these issues altogether, but it will take a first step in addressing them.

Since we want to design a more sensible language that does not have the idiosyncracies of JavaScript and that handles the network communication for us, we now look for a basis for our language. Murphy's research shows us that a language that uses a modal type system can handle resources in a distributed program elegantly.[18] This thesis will explore what happens when we restrict the distributed system to two computers: client, i.e. user of the web program, and server. As opposed to Murphy's implementation of ML5, which is written in Standard ML, we will attempt to write the entire compiler in a proof assistant

and prove the static correctness of each step. Compilation process from our initial language ML5 to JavaScript will be a series of simple type-directed conversions. Each step should have a specific purpose, and we should be able prove certain properties about the compilation at each step.

Murphy's thesis includes Twelf[22] formalizations of type preservation (static correctness) for the first two steps of compilation, CPS and closure conversion. We formalize the type preservation of two additional steps, lambda-lifting and monomorphization of valid values. Additionally, we give a definition and type system for a fragment of JavaScript, which is used as the target of code generation, and a partial implementation of conversion from the lifted monomorphic language to it. This conversion supports the functional programming constructs: the key challenge of this is explicitly dividing the tierless program into a client portion that depends only on client variables, and similarly for the server; there are other more mundane issues, such as converting sums (which JavaScript does not support) to products with a tag field. It does not yet support the modal and communication constructs. In addition to verifying the static correctness of these further stages of the compilation pipeline, the present work uses a different proof assistant than Murphy's (Agda[19] instead of Twelf), which requires a different style of coding (functional rather than logic programming), and a different representation of variable binding (well-scoped de Bruijn indices rather than higher-order abstract syntax). While the de Bruijn form requires a number of tedious weakening lemmas, it also simplifies closure conversion, because we can directly represent the restriction that a closure is closed by modifying the context of a term, rather than using the auxiliary check on each variable that is used in Twelf. Further, we found the de Bruijn form to be well-suited to lambda-lifting

(computing a list of declarations along with a term inside of them), monomorphization (splitting valid assumptions into a pair of assumptions, one for each world), and conversion to JavaScript (slicing the programs into two programs in two different portions of the context).

The proof assistant Agda[1] that we use is a dependently typed functional programming language with Haskell-like syntax, and this thesis will not go into detail about explaining the language.[2]

Our final program[3] consists of ≈ 3800 lines of executable Agda code. It currently does not have a parser, so the ML5 code has to be written in Agda, using the abstract syntax tree (AST) of ML5. However the task is not as daunting as it sounds, because of the mixfix variable naming in Agda, a program written in

---

[1] We are using Agda 2.5.2 and the Agda standard library 0.12.

[2] Since we will not go into detail about Agda or dependent typing, one crucial concept we should remember is that types can be values in a dependently typed language, just like functions are values in a functional language. Therefore a type will also belong to a type. The type of types in Agda is called Set, which will come up often in this thesis. However, not every Agda type belong to the type Set. Since we will not use this distinction, we will omit the explanation. For more information, you can read about Girard's paradox.

A syntactic reminder about Agda is that it allows mixfix naming of variables. An underscore character in a name stands for an argument; in fact we will define the ternary conditional operator as ‘if_‘then_‘else_ in ML5.

Another feature of Agda is that it allows implicit arguments in functions. If you see an argument that is in curly braces, you do not have to provide its value during the function call. If you do not want to write the type of the implicit argument either , you can write it in the beginning of the type as $f : \forall \{a\ b\ c\} \rightarrow \ldots$ Implicit arguments are used in cases where they can easily be inferred. If we were to prove that zero is less than or equal to all natural numbers, we do not have to write natural number as an explicit argument. We can state the lemma as $pf : \forall \{n\} \rightarrow 0 \leqslant n$.

[3] The Agda source code is available at http://github.com/joom/modal

the AST does not look that different from any other functional language. For example, a simple program that alerts a string on the browser looks like this:

```
program1 : [] ⊢5 ‘Unit < client >
program1 = ‘prim ‘alert ‘in
  (‘ ‘val (‘v "alert" (here refl)) · ‘val (‘string "hello world"))
```

While the source language would look like this:

```
prim alert in (alert "hello world")
```

A more complex program, that prompts the user to enter a string, and then writes the input the screen looks like this:

```
program2 : [] ⊢5 ‘Unit < client >
program2 = ‘prim ‘alert ‘in ‘prim ‘prompt ‘in
  (‘ ‘val (‘v "alert" (there (here refl))) ·
    (‘ ‘val (‘v "prompt" (here refl)) · ‘val (‘string "Write something")))
```

In the source language, this one would look like:

```
prim alert in (prim prompt in (alert (prompt "Write something")))
```

Both of these programs actually compile to JavaScript and run successfully in browser.

Starting from ML5, our compiler has 5 conversion steps: continuation-passing style, closure conversion, lambda lifting, monomorphization, and JavaScript. Currently all the steps are entirely completed except the conversion from the monomorphic language to JavaScript. Currently conversions for base types and function calls work properly, and there is partially working network communication between the client and the server. It is also worth noting that our verification only

proves static correctness by implementing a type-preserving compiler, it does not prove anything about operational semantics.

Now let's go over some aspects of modal logic, so that we can understand the modal type system we will use to organize our web programs.

2. Background

In non-modal propositional logic, certain kinds of notations for inference rules obscure the distinction between a proposition and a judgment. Consider the following conjunction intro rule:

$$\frac{A \qquad B}{A \wedge B}$$

Now are A, B and $A \wedge B$ in this rule propositions or judgments? Syntactically they are propositions, but if we intend to say "$A \wedge B$ is true, if A is true and B is true." then they are in fact judgments. The ambiguity about the notions of proposition and judgment can be removed by adopting a new notation for judgments; we would now write "A true" or "$\vdash A$" for a judgment, instead of just "A".

This notation and the inference rule above do not cover the case in which a proposition depends on other propositions, i.e. when we want to say "A follows from Γ". We accept "$\Gamma \vdash A$" as the default notation such a judgment. If we apply these changes to our inference rules, we would get

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

The distinction between a proposition and judgment does not come up often when we think about non-modal logic, so this distinction might seem like a nuisance. However it will be our gateway to understanding our modal logic. [21]

It should be clarified if an incorrect or unproved judgment is still a judgment. In other words, is the word judgment a way to express truth or is it just a form?

In this regard, I will follow Martin-Löf's fourfold terminology[17]: judgment and proposition are "stripped of [their] epistemic force", they describe the state that those concepts have before "[they have] been proved or become known". On the other hand, the terms "evident judgment" and "true proposition" imply that there is a proof.

2.1. Modal logic. Modal logic is a broad field that includes various kinds of logic that deal with relational structures that have different perspectives of truth. [4, 18] For our purposes, we only want to examine intuitionistic modal logic with explicit worlds.

We call these perspectives of truth, "possible worlds". Each world holds a possibly different set of truths. It is possible for a proposition to be true in one world and false in another.

To illustrate the concept, let's think of a prison that has cells, and each correspond to a world in our modal logic. Suppose in each cell, there is a person who is locked inside. Alice is in a cell with a window, while Bob is in a windowless one. Alice can look outside and learn that it is sunny, however Bob would not be able to do that. In Alice's room you have proof of the nice weather, but in Bob's room you do not.

Let's express this in modal logic. Let A be the proposition that says it is sunny, and w and w' be the worlds of Alice and Bob respectively. We cannot simply say "A true", because we did not specify in which cell that is true. We should say "A is true in Alice's world", for which we will use the notation "A <w>".[4]

Alice, Bob and others in different cells have a warden, whose name is Walter. Walter provides communication among everyone. Alice can take a photo of outside

[4] Notice that A is the proposition here, and A <w> is the entire judgment. The proposition $A \wedge B$ <w> should be read $(A \wedge B)$ <w>, because $A \wedge (B$<w>) does not make sense.

and send it in an envelope to Bob through Walter. Now Bob also has a proof that it is sunny, and he can use it later.

From a modal logic perspective, it matters to regulate who can send envelopes to whom. If we call our set of prisoners, i.e. worlds, W, then this regulation is achieved by a relation $R \subseteq W \times W$.

The properties of the relation R defines the kind of logic we have. The logic with the relation that is reflexive, transitive and symmetric, i.e. an equivalence relation, is called **IS5**.[20] For our purposes we will deal with the case that R is a full relation $R = W \times W$, in other words every world will be accessible from any other. We call this relation **IS5**$^{\cup}$.[18]

In Figure 1, we state the axioms and inference rules of IS5$^{\cup}$ for the connectives that are familiar from non-modal propositional logic, namely $\top, \bot, \wedge, \vee$ and $\supset$ (reads "implies", $\Rightarrow$ is another notation for it), as presented by Murphy.[18]

FIGURE 1. Axioms and inference rules of IS5$^{\cup}$.

$$\frac{}{\Gamma, A\langle \mathrm{w}\rangle \vdash A\langle \mathrm{w}\rangle}\ \mathrm{hyp} \qquad \frac{}{\Gamma \vdash \top\langle \mathrm{w}\rangle}\ \top$$

$$\frac{\Gamma \vdash A\langle \mathrm{w}\rangle \qquad \Gamma \vdash B\langle \mathrm{w}\rangle}{\Gamma \vdash A \wedge B\langle \mathrm{w}\rangle}\ \wedge_i$$

$$\frac{\Gamma \vdash A \wedge B\langle \mathrm{w}\rangle}{\Gamma \vdash A\langle \mathrm{w}\rangle}\ \wedge_{e_1} \qquad \frac{\Gamma \vdash A \wedge B\langle \mathrm{w}\rangle}{\Gamma \vdash B\langle \mathrm{w}\rangle}\ \wedge_{e_2}$$

$$\frac{\Gamma \vdash A\langle \mathrm{w}\rangle}{\Gamma \vdash A \vee B\langle \mathrm{w}\rangle}\ \vee_{i_1} \qquad \frac{\Gamma \vdash B\langle \mathrm{w}\rangle}{\Gamma \vdash A \vee B\langle \mathrm{w}\rangle}\ \vee_{i_2}$$

$$\frac{\Gamma \vdash A \vee B\langle \mathrm{w}\rangle \qquad \Gamma, A\langle \mathrm{w}\rangle \vdash C\langle \mathrm{w}\rangle \qquad \Gamma, B\langle \mathrm{w}\rangle \vdash C\langle \mathrm{w}\rangle}{\Gamma \vdash C\langle \mathrm{w}\rangle}\ \vee_e$$

$$\frac{\Gamma, A\langle \mathrm{w}\rangle \vdash B\langle \mathrm{w}\rangle}{\Gamma \vdash A \supset B\langle \mathrm{w}\rangle}\ \supset_i \qquad \frac{\Gamma \vdash A \supset B\langle \mathrm{w}\rangle \qquad \Gamma \vdash A\langle \mathrm{w}\rangle}{\Gamma \vdash B\langle \mathrm{w}\rangle}\ \supset_e$$

This is a good start, but if we want to make more general claims about different worlds, these connectives do not suffice. Therefore we introduce the $\Box$ (reads

"box") and $\Diamond$ (reads "diamond") as new modal connectives. $\Box A$ means A is true for all worlds, and $\Diamond A$ means that A is true for some world.[5] The inference rules for $\Box$ and $\Diamond$ are in Figure 2, as presented by Murphy.[18]

Notice that we are using w and w' for concrete world variables, while ω stands for a world that is quantified.

FIGURE 2. Inference rules for $\Box$ and $\Diamond$ in IS5$^{\cup}$

$$\frac{\Gamma, \omega \text{ world} \vdash A\langle\omega\rangle}{\Gamma \vdash \Box A\langle\text{w}\rangle}\,\Box_i \qquad \frac{\Gamma \vdash \Box A\langle\text{w'}\rangle}{\Gamma \vdash A\langle\text{w}\rangle}\,\Box_e$$

$$\frac{\Gamma \vdash A\langle\text{w'}\rangle}{\Gamma \vdash \Diamond A\langle\text{w}\rangle}\,\Diamond_i \qquad \frac{\Gamma \vdash \Diamond A\langle\text{w'}\rangle \qquad \Gamma, \omega \text{ world}, A\langle\omega\rangle \vdash C\langle\text{w}\rangle}{\Gamma \vdash C\langle\text{w}\rangle}\,\Diamond_e$$

2.1.1. *Hybrid logic and quantifiers.* Even though $\Box$ and $\Diamond$ are introduced as modal connectives, they are not as expressive as we like. Moreover, for any person who is familiar with first-order logic, universal and existential quantifiers ($\forall$ and $\exists$) are more intuitive. For that reason, we will replace them with three different connectives: $\forall, \exists$ and `at`. [15, 16] The inference rules for them are presented in Figure 3, as presented by Murphy.[18]

A `at` w is a proposition that is an internalization of the $A\langle$w$\rangle$ judgment, just like $A \supset B$ is an internalization of the $A \vdash B$ judgment. [18]

$\forall\omega.A$ means "for all worlds ω, the proposition A is true". Similarly, $\exists\omega.A$ means "there exists a world ω such that the proposition A is true". Notice that our propositions now can contain references to worlds. In other words, ω is a bound variable for a world in the proposition A above.

2.2. Lambda 5. We stated the rules of modal logic in the previous section, and now we want to convert each rule to a proof term, and hence define a language

[5] Technically $\Box$ and $\Diamond$ should say "all worlds accessible from the current one", but since we are using IS5$^{\cup}$ and all worlds are accessible from each other, we can directly say "all worlds".

FIGURE 3. Inference rules of hybrid connectives for IS5$^{\cup}$

$$\frac{\Gamma \vdash A\text{<w'>}}{\Gamma \vdash A \texttt{ at } \text{w'<w>}}\ \texttt{at}_i \qquad \frac{\Gamma \vdash A \texttt{ at } \text{w"<w'>} \qquad \Gamma, A\text{<w">} \vdash C\text{<w'>}}{\Gamma \vdash C\text{<w>}}\ \texttt{at}_e$$

$$\frac{\Gamma, \omega \text{ world} \vdash A\text{<}\omega\text{>}}{\Gamma \vdash \forall\omega.A\text{<w>}}\ \forall_i \qquad \frac{\Gamma \vdash \forall\omega.A\text{<w>}}{\Gamma \vdash [\text{w'}/\omega]A\text{<w>}}\ \forall_e$$

$$\frac{\Gamma \vdash [w'/\omega]A\text{<w>}}{\Gamma \vdash \exists\omega.A\text{<w>}}\ \exists_i$$

$$\frac{\Gamma \vdash \exists\omega.A\text{<w'>} \qquad \Gamma, \omega \text{ world}, A\text{<}\omega\text{>} \vdash C\text{<w>} \qquad \omega \notin FV(C)}{\Gamma \vdash C\text{<w>}}\ \exists_e$$

that we will call Lambda 5.[6] 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. In simpler words, modal propositions will be types in Lambda 5, and proof trees will be Lambda 5 expressions. Figure 4 shows the proof terms of Lambda 5, as presented by Murphy[18, 15]

2.2.1. *Mobility.* If we go back to the prison analogy, it is clear that the rules in Figure 1 are not enough to provide communication between different worlds. However we have to think about what kinds of proofs Alice can pass onto Bob on a paper. Given that Alice has a window in her cell and Bob does not, can Alice instruct Bob how to look up the weather by himself? The answer is no, whatever Alice writes down, Bob will always have to ask someone else. So we

[6]We should note that our Lambda 5 definition is different from the one defined by Murphy[18]. Our definition includes $\forall$ and $\exists$ for worlds, while Murphy uses $\Box$ and $\Diamond$ instead. Since this thesis is more about the compilation process than logic itself, we choose to fast-forward to quantifiers. However, our inference rules for $\forall$ and $\exists$ are borrowed from the MinML5 definition in Murphy's work. We are just using the name Lambda 5 for a different definition than he does.

FIGURE 4. Proof terms of Lambda 5

$$\frac{}{\Gamma, x : A\text{<w>} \vdash \mathsf{v}\ x : A\text{<w>}}\ \text{hyp} \qquad \frac{}{\Gamma \vdash \mathsf{tt} : \top\text{<w>}}\ \top$$

$$\frac{\Gamma \vdash M : A\text{<w>} \qquad \Gamma \vdash N : B\text{<w>}}{\Gamma \vdash (M,N) : A \wedge B\text{<w>}}\ \wedge_i$$

$$\frac{\Gamma \vdash M : A \wedge B\text{<w>}}{\Gamma \vdash \mathsf{fst}\ M : A\text{<w>}}\ \wedge_{e_1} \qquad \frac{\Gamma \vdash M : A \wedge B\text{<w>}}{\Gamma \vdash \mathsf{snd}\ M : B\text{<w>}}\ \wedge_{e_2}$$

$$\frac{\Gamma \vdash M : A\text{<w>}}{\Gamma \vdash \mathsf{inl}\ M : A \vee B\text{<w>}}\ \vee_{i_1} \qquad \frac{\Gamma \vdash M : B\text{<w>}}{\Gamma \vdash \mathsf{inr}\ M : A \vee B\text{<w>}}\ \vee_{i_2}$$

$$\frac{\Gamma \vdash M : A \vee B\text{<w>} \qquad \Gamma, x : A\text{<w>} \vdash N_1 : C\text{<w>} \qquad \Gamma, y : B\text{<w>} \vdash N_2 : C\text{<w>}}{\Gamma \vdash \mathsf{case}\ M\ \mathsf{of\ inl}\ x \Rightarrow N_1\ \mathsf{inr}\ y \Rightarrow N_2 : C\text{<w>}}\ \vee_e$$

$$\frac{\Gamma, x : A\text{<w>} \vdash M : B\text{<w>}}{\Gamma \vdash \lambda x.M\ : A \supset B\text{<w>}}\ \supset_i \qquad \frac{\Gamma \vdash M : A \supset B\text{<w>} \qquad \Gamma \vdash N : A\text{<w>}}{\Gamma \vdash M\ N : B\text{<w>}}\ \supset_e$$

$$\frac{\Gamma \vdash N : N : A\text{<w'>}}{\Gamma \vdash \mathsf{held}\ N : A\ \mathtt{at}\ \text{w'<w>}}\ \mathtt{at}_i$$

$$\frac{\Gamma \vdash M : A\ \mathtt{at}\ \text{w"<w'>} \qquad \Gamma, x : A\text{<w">} \vdash N : C\text{<w'>}}{\Gamma \vdash \mathsf{leta}\ x = M\ \mathbf{in}\ N : C\text{<w>}}\ \mathtt{at}_e$$

$$\frac{\Gamma, \omega\ \text{world} \vdash v : A\text{<}\omega\text{>}}{\Gamma \vdash \Lambda\omega.v\ :\ \forall\omega.A\text{<w>}}\ \forall_i \qquad \frac{\Gamma \vdash M : \forall\omega.A\text{<w>}}{\Gamma \vdash M\langle\text{w'}\rangle : [^{\text{w'}}/_{\omega}]A\text{<w>}}\ \forall_e$$

$$\frac{\Gamma \vdash v : [^{w'}/_{\omega}]A\text{<w>}}{\Gamma \vdash \mathsf{wpair}\ \text{w}\ v : \exists\omega.A\text{<w>}}\ \exists_i$$

$$\frac{\Gamma \vdash M : \exists\omega.A\text{<w>} \qquad \Gamma, \omega\ \text{world}, x : A\text{<}\omega\text{>} \vdash N : C\text{<w>} \qquad \omega \notin FV(C)}{\Gamma \vdash \mathsf{unpack}\ x = M\ \mathbf{in}\ N : C\text{<w>}}\ \exists_e$$

discover that communication between cells are restricted by nature, because they can only pass notes on a paper, and they are confined to their cells. We will later call the process of writing down data "marshaling".

The restriction they have is communication. They cannot write down information that contains communication from their cell. Therefore we define a concept

of mobility in Figure 5 and then a communication inference rule in Figure 6, as presented by Murphy.[18]

FIGURE 5. Mobility judgment for Lambda 5

$$\frac{}{\top \text{ mobile}}\,\top_m \qquad \frac{}{A \texttt{ at w mobile}}\,\texttt{at}_m$$

$$\frac{A \text{ mobile} \qquad B \text{ mobile}}{A \wedge B \text{ mobile}}\,\wedge_m \qquad \frac{A \text{ mobile} \qquad B \text{ mobile}}{A \vee B \text{ mobile}}\,\vee_m$$

$$\frac{A \text{ mobile}}{\forall\omega.A \text{ mobile}}\,\forall_m \qquad \frac{A \text{ mobile}}{\exists\omega.A \text{ mobile}}\,\exists_m$$

FIGURE 6. Communication inference rule for Lambda 5

$$\frac{A \text{ mobile} \qquad \Gamma \vdash N : A\text{<w'>}}{\Gamma \vdash \textsf{get } N : A\text{<w>}}\,\text{get}$$

2.2.2. *Validity.* Currently we only have one possible judgment that is specifically for a single world. Later these will correspond to programs that run in a single world. We should consider that many functions we use in programs are shared by the client and server, such as simple library functions. [18]

One can argue that the $\Box$ connective (or $\forall$ for worlds) would be enough to cover this purpose. However we have to remember that even in that case the judgment will be in a specific world. Say we have a term $M : \forall\omega.A\text{<w>}$. This will still be a term in the world w, which means if we want to use it in a different world we will have to move it explicitly, as we saw in subsubsection 2.2.1.

Remember that in our judgment structure, before the $\vdash$, we have smaller judgments (i.e. hypotheses) that look like $x : A\text{<w>}$. Now to solve the problem we described above, we will introduce another judgment that describes a value that is valid in all worlds. We will denote this judgment as $u \sim \omega.A$, where u is the name of the variable, and A is a proposition in which the world ω is bound.

In subsubsection 2.1.1 we mentioned that some judgments can be internalized as propositions, such as $A\text{<w>}$ to A `at` w and $A \vdash B$ to $A \supset B$. This raises the question of what an internalization of $u \sim \omega.A$ would look like. Ergo we define a $\Box$-like proposition named ⌘ (read "shamrock"). [15]

FIGURE 7. Validity rules for in Lambda 5, as presented by Murphy.[18]

$$\frac{\Gamma, \omega \text{ world} \vdash M : A\text{<}\omega\text{>}}{\Gamma \vdash \mathsf{sham}\ \omega.M : ⌘_\omega A\text{<w>}}\ ⌘_i$$

$$\frac{\Gamma \vdash M : ⌘_\omega A\text{<w'>} \qquad \Gamma, u \sim \omega.A \vdash N : C\text{<w'>}}{\Gamma \vdash \mathsf{letsham}\ u = M\ \mathbf{in}\ N : C\text{<w>}}\ ⌘_e$$

$$\frac{}{\Gamma, u \sim \omega.A \vdash \mathsf{vval}\ u : [^w/_\omega]A\text{<w>}}\ \text{vhyp}$$

$$\frac{A \text{ mobile} \qquad \Gamma \vdash M : A\text{<w>} \qquad \Gamma, u \sim \omega.A \vdash N : C\text{<w>}}{\Gamma \vdash \mathsf{put}\ u = M\ \mathbf{in}\ N : C\text{<w>}}\ \text{put}$$

$$\frac{}{⌘A \text{ mobile}}\ ⌘_m$$

3. Type-directed translation

We mentioned in section 1 that our compiler has 5 conversion steps: continuation-passing style, closure conversion, lambda lifting, monomorphization, and JavaScript.

We will start this section by describing ML5, which is more or less an Agda formalization of Lambda 5 that we described in subsection 2.2. Our next task will be to define a similar language that respects continuation-passing style, and convert from ML5 to the new one. Considering that most actions in JavaScript are run through callbacks[7], this process is necessary to move us closer to JavaScript,

[7]A callback in JavaScript is not a syntactic construct, it is rather a widely-used pattern. When there is a computation that can take long, or will have to do communication, the common usage is to pass it a function as an argument, which will be called when the computation is finished. Because of this, programmers often end up with nested anonymous function that are

our final target language. Then 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. Only after that we can hoist the functions, i.e. lambda lifting. Finally, before conversion to JavaScript, we have to monomorphize valid values into values in specific worlds.

Through every step, we specify a way, namely convertType, that a type will be converted into the next language and show that our conversion obeys the restrictions of that. In other words, we are implementing a type-preserving compiler using the Agda proof assistant and verifying static correctness.[8]

Now that we have a broader view about the entire compilation process, let's define what a world is in our languages.

```
data World : Set where
  client : World
  server : World
```

We define World as a data type that only has two values. Note that our approach is different from Licata and Harper[15], they use Agda's **postulate** keyword to assume that there is a type World and it has at least two values client and server, leaving the door open for other possible values. However our definition shows that there is no other possible value for the type World, which means we can do induction on it if necessary, and show that world equality is decidable. We will use the same definition of World in all of our languages.

hard to read, also called "callback hell". Since we are generating code that is not meant to be read, we can overlook that problem.

[8]As we said in the introduction, we are not proving anything about operational semantics.

3.1. ML5. In the previous section we presented a propositional modal logic and the proof terms that correspond to that logic. In this section, our goal is to formalize this language and the compilation process to the monomorphic language.

First, we should define each type in our initial language:

```
data Type : Set where
  ‘Int ‘Bool ‘Unit ‘String : Type
  ‘_⇒_ ‘_×_ ‘_⊎_ : Type → Type → Type
  ‘_at_ : Type → World → Type
  ‘⌘ : (World → Type) → Type
  ‘∀ ‘∃ : (World → Type) → Type
```

Then we have to define the building block judgments that we will use in the "... ⊢ ..." kind of judgment. We should start with what can come before the ⊢. We will call this small judgment a hypothesis.

```
data Hyp : Set where
  _∶_<_> : (x : Id) (τ : Type) (w : World) → Hyp
  _∼_ : (u : Id) → (World → Type) → Hyp
```

We have two kinds of hypothesis judgments, the first is the judgment for a specific world that we described in subsection 2.1.[9] and the second is the valid value judgment we defined in subsubsection 2.2.2. It is worth mentioning that Agda formalization of a valid value takes a function World → Type, we have to apply the a world to this function if we want to actually use the type.[15] Using a function for this saves us from the trouble of having to define substitution ourselves. We will also define Context = List Hyp.

We will call what comes after the small judgment ⊢ a conclusion.

[9]For convenience we defined Id = String, where String is the Agda type for strings.

```
data Conc : Set where
  _<_> : (τ : Type) (w : World) → Conc
  ↓_<_> : (τ : Type) (w : World) → Conc
```

We have two kinds of conclusion: we will call the first an expression and the second, a value. In our definition, an expression is a term that potentially requires network communication for evaluation. A value, on the other hand, does not.[10] Therefore an expression conclusion will look like $\tau < w >$, and a value conclusion will look like $\downarrow \tau < w >$.

Before defining the terms of ML5, let's define the mobility judgment in ML5 so that we can use it in the terms.

```
data _mobile : Type → Set where
  'Bool^m : 'Bool mobile
  'Int^m : 'Int mobile
  'Unit^m : 'Unit mobile
  'String^m : 'String mobile
  '_at^m_ : ∀ {A w} → (' A at w) mobile
  '_×^m_ : ∀ {A B} → A mobile → B mobile → (' A × B) mobile
  '_⊎^m_ : ∀ {A B} → A mobile → B mobile → (' A ⊎ B) mobile
  '∀^m : ∀ {A} → A mobile → ('∀ (λ _ → A)) mobile
  '∃^m : ∀ {A} → A mobile → ('∃ (λ _ → A)) mobile
  '⌘^m : ∀ {A} → ('⌘ (λ _ → A)) mobile
```

[10] It is worth noting that what we call a value is not literally a value. Intuitively most readers think that a value is something that cannot be further evaluated. However in our definition $4 + 5$ is a value. What we mean by the word value is in fact a pure computation, one that does not require any communication. That being said, we will keep using the term "value" to be consistent with Murphy[18], and Licata and Harper.[15]

Now that we have defined mobility as a direct formalization of Figure 5, let's define the notion of terms in ML5 and terms of the base types.

```
data _⊢_ (Γ : Context) : Conc → Set where
  'tt : ∀ {w} → Γ ⊢ ↓ 'Unit < w >
  'string : ∀ {w} → Data.String.String → Γ ⊢ ↓ 'String < w >
  'true : ∀ {w} → Γ ⊢ ↓ 'Bool < w >
  'false : ∀ {w} → Γ ⊢ ↓ 'Bool < w >
  '_∧_ : ∀ {w} → Γ ⊢ ↓ 'Bool < w > → Γ ⊢ ↓ 'Bool < w > → Γ ⊢ ↓ 'Bool < w >
  '_∨_ : ∀ {w} → Γ ⊢ ↓ 'Bool < w > → Γ ⊢ ↓ 'Bool < w > → Γ ⊢ ↓ 'Bool < w >
  '¬_ : ∀ {w} → Γ ⊢ ↓ 'Bool < w > → Γ ⊢ ↓ 'Bool < w >
  'if_'then_'else_ : ∀ {τ w} → Γ ⊢ 'Bool < w >
    → Γ ⊢ τ < w > → Γ ⊢ τ < w > → Γ ⊢ τ < w >
  'n_ : ∀ {w} → ℤ → Γ ⊢ ↓ 'Int < w >
  '_⩽_ : ∀ {w} → Γ ⊢ ↓ 'Int < w > → Γ ⊢ ↓ 'Int < w > → Γ ⊢ ↓ 'Bool < w >
  '_+_ : ∀ {w} → Γ ⊢ ↓ 'Int < w > → Γ ⊢ ↓ 'Int < w > → Γ ⊢ ↓ 'Int < w >
  '_*_ : ∀ {w} → Γ ⊢ ↓ 'Int < w > → Γ ⊢ ↓ 'Int < w > → Γ ⊢ ↓ 'Int < w >
```

We define _⊢_ to be the type of terms, similar to our usage in subsection 2.2. Then we define literals and some operators for the base types 'Unit, 'String, 'Bool and 'Int. In order to make the language more practical, one can expand this to new base types and operators, such as floating numbers, hexadecimals etc., but what we have here is enough for a proof of concept.

```
'v : ∀ {τ w} → (x : Id) → x ∶ τ < w > ∈ Γ → Γ ⊢ ↓ τ < w >
'vval : ∀ {w C} → (u : Id) → u ∼ C ∈ Γ → Γ ⊢ ↓ C w < w >
```

We are defining two values to use variables that are already in the context.[11] The first is for the judgment $\mathsf{x} \overset{\circ}{:} \tau < \mathsf{w} >$, which refers to values that are in a specific world. The second is for the judgment $\mathsf{u} \sim \mathsf{C}$, which refers to valid values. In both cases, our values require a proof that the variable we are using is actually in the context.

$$
\begin{aligned}
&\text{`}\lambda\_\overset{\circ}{:}\_\Rightarrow\_ \;:\; \forall\,\{\tau\;\mathsf{w}\} \to (\mathsf{x} \;:\; \mathsf{Id})\;(\sigma \;:\; \mathsf{Type}) \\
&\quad \to (\mathsf{x} \overset{\circ}{:} \sigma < \mathsf{w} > :: \Gamma) \vdash \tau < \mathsf{w} > \\
&\quad \to \Gamma \vdash \downarrow (\text{`}\,\sigma \Rightarrow \tau) < \mathsf{w} > \\
&\text{`}\_\cdot\_ \;:\; \forall\,\{\tau\;\sigma\;\mathsf{w}\} \to \Gamma \vdash (\text{`}\,\tau \Rightarrow \sigma) < \mathsf{w} > \to \Gamma \vdash \tau < \mathsf{w} > \to \Gamma \vdash \sigma < \mathsf{w} >
\end{aligned}
$$

We define lambda functions and function application. A lambda function takes the arguments name and type, and a function body that now contains the argument as well. A lambda function is a value by itself, because it cannot be further reduced and therefore its evaluation does not require network connection. However, notice that the function body can require communication, therefore so can a function application. For that reason, function application is an expression.

$$
\begin{aligned}
&\text{`}\_,\_ \;:\; \forall\,\{\tau\;\sigma\;\mathsf{w}\} \to \Gamma \vdash \downarrow \tau < \mathsf{w} > \to \Gamma \vdash \downarrow \sigma < \mathsf{w} > \to \Gamma \vdash \downarrow (\text{`}\,\tau \times \sigma) < \mathsf{w} > \\
&\text{`}\mathsf{fst} \;:\; \forall\,\{\tau\;\sigma\;\mathsf{w}\} \to \Gamma \vdash (\text{`}\,\tau \times \sigma) < \mathsf{w} > \to \Gamma \vdash \tau < \mathsf{w} > \\
&\text{`}\mathsf{snd} \;:\; \forall\,\{\tau\;\sigma\;\mathsf{w}\} \to \Gamma \vdash (\text{`}\,\tau \times \sigma) < \mathsf{w} > \to \Gamma \vdash \sigma < \mathsf{w} > \\
&\text{`}\mathsf{inl}\_\text{`}\mathsf{as}\_ \;:\; \forall\,\{\tau\;\mathsf{w}\} \to \Gamma \vdash \downarrow \tau < \mathsf{w} > \to (\sigma \;:\; \mathsf{Type}) \to \Gamma \vdash \downarrow (\text{`}\,\tau \uplus \sigma) < \mathsf{w} > \\
&\text{`}\mathsf{inr}\_\text{`}\mathsf{as}\_ \;:\; \forall\,\{\sigma\;\mathsf{w}\} \to \Gamma \vdash \downarrow \sigma < \mathsf{w} > \to (\tau \;:\; \mathsf{Type}) \to \Gamma \vdash \downarrow (\text{`}\,\tau \uplus \sigma) < \mathsf{w} >
\end{aligned}
$$

[11] We are using the definition of the type $\_\in\_$ for list membership in the Agda standard library. Suppose we are looking at $\mathsf{x} \in \mathsf{xs}$. It has two constructors, here is applicable if the x is the head of the list. The other constructor there is a way to skip one element in the list; if you know $\mathsf{x} \in \mathsf{xs}$, then you know $\mathsf{x} \in \mathsf{y} :: \mathsf{xs}$ for all y. The idea behind this type is the same as De Bruijn indices. If you consider the natural number type with the constructors zero and suc, here corresponds to zero and suc corresponds to there.

```
‘case_‘of_⇒_||_⇒_ : ∀ {τ σ υ w} → Γ ⊢ (‘ τ ⊎ σ) < w >
  → (x : Id) → (x ⸬ τ < w > :: Γ) ⊢ υ < w >
  → (y : Id) → (y ⸬ σ < w > :: Γ) ⊢ υ < w >
  → Γ ⊢ υ < w >
```

Then we define terms for the product type ‘_×_, which corresponds to the logical connective $\wedge$, and the sum type ‘_⊎_, which corresponds to the logical connective $\vee$. We define introduction and elimination terms for both.

```
‘hold : ∀ {τ w w’} → Γ ⊢ ↓ τ < w’ > → Γ ⊢ ↓ (‘ τ at w’) < w >
‘leta_‘=_‘in_ : ∀ {τ σ w w’} → (x : Id) → Γ ⊢ (‘ τ at w’) < w >
  → ((x ⸬ τ < w’ >) :: Γ) ⊢ σ < w >
  → Γ ⊢ σ < w >
‘sham : ∀ {w} {A : World → Type}
  → ((ω : World) → Γ ⊢ ↓ (A ω) < ω >) → Γ ⊢ ↓ ‘⌘ A < w >
‘letsham_‘=_‘in_ : ∀ {σ w} {A : World → Type} → (u : Id) → Γ ⊢ ‘⌘ A < w >
  → (u ∼ A :: Γ) ⊢ σ < w > → Γ ⊢ σ < w >
‘put : ∀ {τ σ w} {m : τ mobile} (u : Id) → Γ ⊢ τ < w >
  → ((u ∼ (λ _ → τ)) :: Γ) ⊢ σ < w >
  → Γ ⊢ σ < w >
```

We define introduction and elimination terms for the at and ⌘ connectives we defined in subsubsection 2.1.1 and subsubsection 2.2.2. Our terms here are direct formalizations of the inference rules $\mathsf{at}_i, \mathsf{at}_e$, $⌘_i$, $⌘_e$ and put, respectively, defined in Figure 4 and Figure 7.

```
‘Λ : ∀ {w} {A : World → Type} → ((ω : World) → Γ ⊢ ↓ A ω < w >) → Γ ⊢ ↓ ‘∀ A < w >
_⟨_⟩ : ∀ {w} {A : World → Type} → Γ ⊢ ‘∀ A < w > → (ω : World) → Γ ⊢ (A ω) < w >
‘wpair : ∀ {w} {A : World → Type} (ω : World) → Γ ⊢ ↓ A ω < w > → Γ ⊢ ↓ ‘∃ A < w >
```

```
‘unpack_‘=_‘in_ : ∀ {w τ} {A : World → Type} (x : Id) → Γ ⊢ ‘∃ A < w >
  → ((ω : World) → ((x ⦂ A ω < w >) :: Γ) ⊢ τ < w >) → Γ ⊢ τ < w >
```

We define introduction and elimination terms for the $\forall$ and $\exists$ connectives we defined in subsubsection 2.1.1. Our terms are direct formalizations of the inference rules $\forall_i, \forall_e, \exists_i, \exists_e$, respectively, defined in Figure 4.

```
‘get : ∀ {τ w w'} {m : τ mobile} → Γ ⊢ τ < w' > → Γ ⊢ τ < w >
‘val : ∀ {τ w} → Γ ⊢ ↓ τ < w > → Γ ⊢ τ < w >
‘prim_‘in_ : ∀ {h w σ} (x : Prim h) → (h :: Γ) ⊢ σ < w > → Γ ⊢ σ < w >
```

We define the most important term in ML5, get, in order to move a mobile expression from one world to another. It is a term after the inference rule we defined in Figure 6. get will often be an interesting case during our type-directed conversions; keeping an eye out for it will help us understand the compiler better.

We define the term val to inject values into expressions.[15] Since an expression is a term that potentially requires communication, a value can also be an expression.

We define an expression to generalize primitive functions we will have in our language. We will define a type Prim : Hyp → Set that will contain all of our primitives, instead of adding new terms to the type _⊢_. We will not go over the primitive much throughout the conversion, but our Prim type in ML5 is as follows:

```
data Prim : Hyp → Set where
  ‘alert : Prim ("alert" ⦂ ‘ ‘String ⇒ ‘Unit < client >)
  ‘write : Prim ("write" ⦂ ‘ ‘String ⇒ ‘Unit < client >)
  ‘version : Prim ("version" ⦂ ‘String < server >)
  ‘log : Prim ("log" ∼ (λ _ → ‘ ‘String ⇒ ‘Unit))
```

```
‘prompt : Prim ("prompt" ⨟ ‘ ‘String ⇒ ‘String < client >)
‘readFile : Prim ("readFile" ⨟ ‘ ‘String ⇒ ‘String < server >)
```

Observe that having valid value primitives also makes sense in some cases such as ‘log, because logging to console makes sense in both client and server.

With that, we conclude the definition of ML5.

3.2. CPS. Currently we have an ML5 program whose control flow is unspecified. We need a language that can represent the entire control and data flow explicitly.[2] We will achieve this by representing the control stack in a function that will be called the continuation function K.[14] The idea behind a continuation function is not far from a JavaScript callback; both are functions that are called when a certain computation is completed. Since nothing is returned and the rest of the computation happens through the continuation function, our resulting program in CPS will be an expression without a type, we will call this a continuation expression, ⋆< w >.

```
data Conc : Set where
  ⋆<_> : (w : World) → Conc
  ↓_<_> : (τ : Type) (w : World) → Conc
```

In the CPS language, we have a different conclusion judgments. ⋆<_> is the continuation expression that denotes a computation that calls the continuation function inside. On the other hand, our value judgment ↓_<_> remains the same.

We will also replace the function type ‘_⇒_ with an alternative that respects the continuations.

```
‘_cont : Type → Type
```

As we mentioned above, expression evaluation does not return anything anymore, it rather calls a continuation function with the computed value. In that case, a function does not have a return type; it only has an argument type.

The value terms in ML5 stay the same in the CPS language, however the expressions all have to become continuation expressions. Let's go over the new terms in our language.

$$
\begin{aligned}
&\text{`if\_`then\_`else\_} \;:\; \forall\,\{w\} \to \Gamma \vdash \downarrow\,(\text{`Bool} < w >) \\
&\quad \to \Gamma \vdash \star < w > \\
&\quad \to \Gamma \vdash \star < w > \\
&\quad \to \Gamma \vdash \star < w > \\
&\text{`letcase\_,\_`=\_`in\_`or\_} \;:\; \forall\,\{\tau\ \sigma\ w\} \to (x\ y\ :\ \mathsf{Id}) \\
&\quad \to \Gamma \vdash \downarrow\,(\text{`}\tau \uplus \sigma) < w > \\
&\quad \to ((x \wr \tau < w >) :: \Gamma) \vdash \star < w > \\
&\quad \to ((y \wr \sigma < w >) :: \Gamma) \vdash \star < w > \\
&\quad \to \Gamma \vdash \star < w > \\
&\text{`let\_`=fst\_`in\_} \;:\; \forall\,\{\tau\ \sigma\ w\} \to (x\ :\ \mathsf{Id}) \\
&\quad \to \Gamma \vdash \downarrow\,(\text{`}\tau \times \sigma) < w > \\
&\quad \to ((x \wr \tau < w >) :: \Gamma) \vdash \star < w > \\
&\quad \to \Gamma \vdash \star < w > \\
&\text{`let\_`=snd\_`in\_} \;:\; \forall\,\{\tau\ \sigma\ w\} \to (x\ :\ \mathsf{Id}) \\
&\quad \to \Gamma \vdash \downarrow\,(\text{`}\tau \times \sigma) < w > \\
&\quad \to ((x \wr \sigma < w >) :: \Gamma) \vdash \star < w > \\
&\quad \to \Gamma \vdash \star < w >
\end{aligned}
$$

The conditional expression and the elimination terms of `_⊎_ and `_×_ become continuation expressions. All expressions in the corresponding terms in ML5 are now $\star < w >$ continuation expressions.

$$
\begin{aligned}
&\text{`leta\_`=\_`in\_} : \forall\,\{\tau\ w\ w'\} \to (x : \mathsf{Id}) \\
&\quad \to \Gamma \vdash \downarrow (\text{`}\,\tau\ \mathsf{at}\ w') < w > \\
&\quad \to ((x \wr \tau < w' >) :: \Gamma) \vdash \star < w > \\
&\quad \to \Gamma \vdash \star < w > \\
&\text{`lets\_`=\_`in\_} : \forall\,\{C\ w\} \to (u : \mathsf{Id}) \\
&\quad \to \Gamma \vdash \downarrow (\text{`}\mathbin{⌘} C) < w > \\
&\quad \to ((u \sim C) :: \Gamma) \vdash \star < w > \\
&\quad \to \Gamma \vdash \star < w > \\
&\text{`put\_`=\_`in\_} : \forall\,\{\tau\ w\}\,\{m : \tau\ \mathsf{mobile}\} \to (u : \mathsf{Id}) \\
&\quad \to \Gamma \vdash \downarrow \tau < w > \\
&\quad \to ((u \sim (\lambda\ \_ \to \tau)) :: \Gamma) \vdash \star < w > \\
&\quad \to \Gamma \vdash \star < w > \\
&\text{`let\_`=\_}\langle\_\rangle\text{`in\_} : \forall\,\{C\ w\} \to (x : \mathsf{Id}) \\
&\quad \to \Gamma \vdash \downarrow \text{`}\forall\ C < w > \\
&\quad \to (w' : \mathsf{World}) \\
&\quad \to ((x \wr C\ w' < w >) :: \Gamma) \vdash \star < w > \\
&\quad \to \Gamma \vdash \star < w > \\
&\text{`let\_=`unpack\_`in\_} : \forall\,\{w\}\,\{A : \mathsf{World} \to \mathsf{Type}\}\ (x : \mathsf{Id}) \\
&\quad \to \Gamma \vdash \downarrow \text{`}\exists\ A < w > \\
&\quad \to ((\omega : \mathsf{World}) \\
&\quad \to ((x \wr A\ \omega < w >) :: \Gamma) \vdash \star < w >) \\
&\quad \to \Gamma \vdash \star < w >
\end{aligned}
$$

Similarly, `put_`=_`in_ and the elimination terms of at, ⌘, ∀ and ∃ become continuation expressions. All expressions in the corresponding terms in ML5 are now $\star <$ w $>$ continuation expressions, but their contexts remain the same.

$$\text{`go}[\_]\_ \;:\; \forall\,\{w\} \to (w' \;:\; \mathsf{World}) \to \Gamma \vdash \star{<}\, w'\, {>} \to \Gamma \vdash \star{<}\, w\, {>}$$

$$\text{`prim}\_\text{`in}\_ \;:\; \forall\,\{h\ w\} \to (x \;:\; \mathsf{Prim}\ h) \to (h :: \Gamma) \vdash \star{<}\, w\, {>} \to \Gamma \vdash \star{<}\, w\, {>}$$

$$\text{`call} \;:\; \forall\,\{\tau\ w\} \to \Gamma \vdash \downarrow\, \text{`}\,\tau\ \mathsf{cont} < w > \to \Gamma \vdash \downarrow \tau < w > \to \Gamma \vdash \star{<}\, w\, {>}$$

$$\text{`halt} \;:\; \forall\,\{w\} \to \Gamma \vdash \star{<}\, w\, {>}$$

Observe that our definition of ‘go differs from its counterpart ‘get in the sense that it does not require mobility, because our expressions no longer have a type that we can check for mobility.[12] Our definition of primitives remain the same except the judgment change. However, notice that function calls are different especially because the function type ‘_⇒_ in ML5 does not exist in the CPS language; it is replaced by ‘_cont. We also define a new term ‘halt, which will be used in the initial continuation function, as marker for the end of the control stack.

This concludes our definition of the CPS language, but before we move on to the conversion process, let's state weakening lemmas for values and continuation expressions:

$$\subseteq\text{-term-lemma} \;:\; \forall\,\{\Gamma\ \Gamma'\ \tau\ w\} \to \Gamma \subseteq \Gamma' \to \Gamma \vdash \downarrow \tau < w > \to \Gamma' \vdash \downarrow \tau < w >$$

$$\subseteq\text{-cont-lemma} \;:\; \forall\,\{\Gamma\ \Gamma'\ w\} \to \Gamma \subseteq \Gamma' \to \Gamma \vdash \star{<}\, w\, {>} \to \Gamma' \vdash \star{<}\, w\, {>}$$

Both proofs are simple inductions.

3.2.1. *Conversion from ML5 to CPS.* During conversion from ML5 to CPS, our initial idea is that each type in ML5 will correspond to another in the CPS language. We define a function to convert ML5 types to CPS types.

$$\mathsf{convertType} \;:\; \mathsf{Type}_5 \to \mathsf{Type}_\times$$

$$\mathsf{convertType}\ \text{`Int} \;=\; \text{`Int}$$

[12] We will fix this by using ‘put_‘=_‘in_, which requires mobility, when we are converting ‘get.

```
convertType ‘Bool = ‘Bool

convertType ‘Unit = ‘Unit

convertType ‘String = ‘String

convertType (‘ τ × σ) = ‘ (convertType τ) × (convertType σ)

convertType (‘ τ ⊎ σ) = ‘ (convertType τ) ⊎ (convertType σ)

convertType (‘ τ at w) = ‘ (convertType τ) at w

convertType (‘⌘ C) = ‘⌘ (λ ω → convertType (C ω))

convertType (‘∀ C) = ‘∀ (λ ω → convertType (C ω))

convertType (‘∃ C) = ‘∃ (λ ω → convertType (C ω))

convertType (‘ τ ⇒ σ) = ‘ (‘ (convertType τ) × (‘ convertType σ cont)) cont
```

This seems like a simple induction except the last case, in which a function is converted into a continuation term that takes a pair of an argument and a callback function for when the return value is ready. We describe the conversion of the lambda term in the interesting conversion cases.

To convert ML5 to the CPS language, we need one function to convert the value type, and another one to convert expressions to the continuation expressions ⋆< w >.

```
convertValue : ∀ {Γ τ w}
  → Γ ⊢₅ ↓ τ < w >
  → (convertCtx Γ) ⊢ₓ ↓ (convertType τ) < w >
convertExpr : ∀ {Γ τ w}
  → (K : ∀ {Γ'} {s' : (convertCtx Γ) ⊆ Γ'}
    → Γ' ⊢ₓ ↓ (convertType τ) < w >
    → Γ' ⊢ₓ ⋆< w >)
  → Γ ⊢₅ τ < w >
  → (convertCtx Γ) ⊢ₓ ⋆< w >
```

The first conversion function's type tells us that a value in ML5 can be converted into a value in the CPS language. The second conversion function's type tells us that if we have an expression in ML5, and if we have a continuation function K that tells us what to do when the expression is evaluated, we can convert the ML5 expression to CPS continuation expression $\star\!<$ w $>$.[13]

It turns out that defining this function with a direct induction on the values and expressions is not straightforward, because of contexts that are hard to reconcile. Therefore, we slightly alter the type of the functions with generalization for all subsets of convertCtx Γ. This way, our function works as an interleaving of the weakening lemma and the conversion function we originally intended.

```
convertValue' : ∀ {Γ Γ' τ w} {s : (convertCtx Γ) ⊆ Γ'}
  → Γ ⊢₅ ↓ τ < w >
  → Γ' ⊢ₓ ↓ (convertType τ) < w >
convertExpr' : ∀ {Γ Γ' τ w}
  → {s : (convertCtx Γ) ⊆ Γ'}
  → (K : ∀ {Γ''} {s' : Γ' ⊆ Γ''}
    → Γ'' ⊢ₓ ↓ (convertType τ) < w >
    → Γ'' ⊢ₓ ⋆< w >)
  → Γ ⊢₅ τ < w >
  → Γ' ⊢ₓ ⋆< w >
```

[13] We are using the word "continuation" for three concepts, so we must be able to distinguish between them before getting into the conversion process. The first usage is a continuation expression, $\star\!<$ w $>$. It corresponds to the expressions in ML5 and it does not have a type for the reasons stated above. The second usage is continuation type, ‘_cont, the replacement of functions in the CPS language. The third usage is continuation function, often named K. This is an Agda-level function, i.e. a function in the meta language.

Since this is the first of the type directed conversions that we are doing, let's look at some uninteresting and interesting cases of this conversion.

Uninteresting cases

Every time we convert one language to another, there will be uninteresting cases that do not change from language to language, however the types require us to fulfill the formalities. In the next conversions, we will often omit the uninteresting cases, but since this is the first conversion we will go over some of them to have a general sense of what we have to do.

```
convertValue' {s = s} (' t ∧ u) = ' (convertValue' {s = s} t) ∧ (convertValue' {s = s} u)
```

The logical "and" operator for booleans is a simple example of a straightforward conversion with recursive calls.[14] We convert the two terms t and u, and then use the logical "and" operator in the CPS language to construct our new term. There are many unary and binary operators like this in our language, and their conversions consist of usage of the same constructor and recursive calls, like the case we have just seen.

```
convertValue' {s = s} ('v x ∈) = 'v x (s (convert∈ ∈))
```

We define the conversion of a variable reference in ML5 to one in CPS. We have the same constructor, however since the context is changed we have to change the proof that the given element is in the new context. We define a simple inductive lemma convert∈ to handle that problem.

```
convert∈ : ∀ {h Γ} → h ∈ Γ → (convertHyp h) ∈ (convertCtx Γ)
convert∈ (here px) = here (cong convertHyp px)
convert∈ (there i) = there (convert∈ i)
```

[14] Notice that {s = s} in the definition is the syntax to name an implicit variable. We want to use the variable about the subset condition.

Now let's go back to the two convertValue'. We have constructors like ‘Λ and ‘sham that take a function from World to a term. Conversion for them will proceed as follows:

```
convertValue' {s = s} (‘Λ C) = ‘Λ (λ ω → convertValue' {s = s} (C ω))
```

Interesting cases

Now we want to handle the conversion cases that actually are of substance.

```
convertValue' {s = s} (‘λ x ⦂ σ ⇒ t) =
  ‘λ (x ++ "_y") ⦂ (‘ (convertType σ) × ‘ _ cont) ⇒
  (‘let x ‘=fst (‘v (x ++ "_y") (here refl)) ‘in
    convertExpr' {s = sub-lemma (there ∘ s)}
      (λ {_} {s'} v →
      ‘let (x ++ "_k") ‘=snd ‘v (x ++ "_y") (s' (there (here refl)))
      ‘in (‘call (‘v (x ++ "_k") (here refl)) (⊆-term-lemma there v))) t)
```

The first and most prominent case we have to handle is the conversion of lambdas. In the definition of convertType we have seen that a function type is converted to a continuation type that takes a pair consisting of the original argument type and a callback function that takes the return type. Therefore the term we are converting is a lambda that gives new names to both elements of the pair it takes as an argument, and calls the second part, which is a callback function, with the first part, which is the initial argument. Notice that the first projection of the pair and the second projection have different contexts, because when the second projection happens the first one is already in the context. Because of this, in order to use the two values together in the same function call, we have to weaken the first projection, i.e. to prove that it is still valid under a greater context.

```
convertExpr' {s = s} K (‘ t · u) =
  convertExpr' {s = s} (λ {_} {s'} v → convertExpr' {s = s' ∘ s}
    (λ {_} {s''} v' → ‘call (⊆-term-lemma s'' v)
      (‘ v' , (‘λ "x" ꟾ _ ⇒ K {s' = there ∘ s'' ∘ s'} (‘v "x" (here refl))))) u) t
```

Since we changed lambda terms so much, we also have to adjust function calls accordingly. What was once a lambda function in ML5 is now a function that takes a pair of the initial argument and a callback function. Therefore during a function call, we should pass a pair of those. The question of what the callback function will do is solved by the continuation function K.

```
convertExpr' {w = w} {s = s} K (‘get {w' = w'} {m = m} t) =
  ‘go[ w' ] (convertExpr' {s = s} (λ {_} {s'} v →
    ‘put_‘=_‘in_ {m = convertMobile m} "u" v
      (‘go[ w ] (K {s' = there ∘ s'} (‘vval "u" (here refl))))) t)
```

We mentioned that ‘go does not contain the notion of mobility, unlike ‘get in ML5. Therefore we need another way of preserving the mobility data we inherit from the ‘get constructor. The only other expression that uses mobility is put, hence we have to use that to ensure mobility. We convert a ‘get to three steps: going to the other world, putting the valid mobile value into the context and then going back to the previous world and using the valid value.

This concludes the conversion of ML5 to the CPS language.

3.3. Closure. Our next goal is to hoist all lambdas in a program to the top, so that we would have names for them, which allows us to use them when we are moving data between the client and the server. Doing this without taking any precautions about bound variables will be disastrous, therefore we first have to convert bound variables to a construct that we can make sure that they will never

become free variables. We do this by creating environment objects before functions and storing them so that later we can refer to the environment objects for the variables that used to depend on a previous definition. We call the combination of the environment object and the lambda, a closure.

Closure language will have two more types different from the CPS language.

```
‘Env : List Hyp → Type
‘Σt[t×[_×t]cont] : Type → Type
```

The first type will be used for the environment objects we described above, and the second type is a hardcoded existential pair we will use to create closures, a combination, i.e. pair, of the environment object and the function.[23]

```
‘λ_⨟_⇒_ : ∀ {w} → (x : Id) (σ : Type)
  → (c : (x ⨟ σ < w > :: []) ⊢ ⋆< w >)
  → Γ ⊢ ↓ (‘ σ cont) < w >
```

We have to update the lambda term to make sure that we are not using anything from the context other than the immediate argument of the lambda we are defining. Notice that the context of the function body c is x ⨟ σ < w > :: [].

Let's start by adding the terms to use the recently introduced type ‘Env.

```
‘buildEnv : ∀ {Δ w} → Δ ⊆ Γ → Γ ⊢ ↓ ‘Env Δ < w >
‘open_‘in_ : ∀ {Δ w} → Γ ⊢ ↓ ‘Env Δ < w > → (Δ ++ Γ) ⊢ ⋆< w > → Γ ⊢ ⋆< w >
```

We will have to add existential pair introduction and elimination rules to our language.

```
‘packΣ : ∀ {σ w} → (τ : Type)
  → Γ ⊢ ↓ (‘ τ × ‘ (‘ σ × τ) cont) < w >
  → Γ ⊢ ↓ ‘Σt[t×[ σ ×t]cont] < w >
```

```
‘let_,_‘=unpack_‘in_ : ∀ {w σ} → (τ : Type) (x : Id)
  → (v : Γ ⊢ ↓ ‘Σt[t×[ σ ×t]cont] < w >)
  → ((x ⦂ ‘ τ × ‘ (‘ σ × τ) cont < w >) :: Γ) ⊢ ⋆< w >
  → Γ ⊢ ⋆< w >
```

Now that we have a hardcoded existential pair type in our language, we can alter the definition of ‘go[_] from our previous language.

```
‘go-cc[_] : ∀ {w} → (w' : World)
  → Data.String.String
    → Γ ⊢ ↓ ‘Σt[t×[ ‘Unit ×t]cont] < w' >
    → Γ ⊢ ⋆< w >
```

That concludes the definition of our closure language. We also define and prove weakening lemmas for continuation expressions and values, with the names ⊆-cont-lemma and ⊆-term-lemma respectively.

3.3.1. *Conversion from CPS to the closure conversion language.* Similar to the CPS conversion, our closure conversion also requires the change of all the types through a function convertType. Most of its definition is the same kind of induction we have seen in subsubsection 3.2.1. The only interesting case is the conversion of ‘_cont, a continuation type becomes an existential pair of the environment object and another continuation type.

```
convertType ‘ τ cont = ‘Σt[t×[ convertType τ ×t]cont]
```

We need a conversion function for values and one for continuations, with the following types.

```
convertValue : ∀ {Γ τ w} → Γ ⊢ₓ ↓ τ < w > → (convertCtx Γ) ⊢ₒ ↓ (convertType τ) < w >
convertCont : ∀ {Γ w} → Γ ⊢ₓ ⋆< w > → (convertCtx Γ) ⊢ₒ ⋆< w >
```

Similar to our previous conversion, closure conversion also looks the direct conversion of contexts and types of the values.

```
convertValue {Γ} {_} {w} (‘λ x ⦂ σ ⇒ t) =
  ‘packΣ (‘Env (convertCtx Γ)) (‘ ‘buildEnv id , ‘λ "p" ⦂ _ ⇒ c)
  where
    t' : convertCtx ((x ⦂ σ < w >) :: Γ) ⊢₀ ⋆< w >
    t' = convertCont t
    c : (("p" ⦂ ‘ convertType σ × ‘Env (convertCtx Γ) < w >) :: []) ⊢₀ ⋆< w >
    c = ‘let "env" ‘=snd ‘v "p" (here refl) ‘in
      ‘open ‘v "env" (here refl) ‘in
      ‘let x ‘=fst ‘v "p" (++ʳ (convertCtx Γ) (there (here refl))) ‘in
      (⊆-cont-lemma (sub-lemma ++ˡ) t')
```

Lambda function is our most essential case in closure conversion. Each function is turned into an existential pair using ‘packΣ, packing the current environment and a new lambda function together. The new function takes a pair of the initial argument and the environment we just built. We expand the function body with expressions to name the environment object, initial argument, and to open up the environment object in the local context. Observe that the argument "p" is the only thing in the context. The lambda function ‘λ "p" ⦂ _ ⇒ c is in fact a closed term, just like we have shown in the defition of the lambda term for the closure language.

Notice that the term t' will be too strong to fit in the last step of our definition, because we defined variables and opened an environment. We have to weaken t' to be able to use it in that spot.

```
convertCont {Γ} (‘call t u) =
  ‘let contextToType Γ , "p" ‘=unpack (convertValue t) ‘in
  ‘let "e" ‘=fst ‘v "p" (here refl) ‘in
  ‘let "f" ‘=snd ‘v "p" (there (here refl)) ‘in
  ‘call (‘v "f" (here refl))
    (‘ ⊆-term-lemma (there ∘ there ∘ there) (convertValue u) , ‘v "e" (there (here refl)))
```

Like CPS conversion, changing the lambda functions requires us to adjust the function applications as well. Since convertValue t is an existential pair, we have to unpack it into an environment and a function. Then we call the function with the pair of the initial argument and the environment we just unpacked in the previous step.

Notice that convertValue u is too strong for the pair that is the argument to the function, because we unpacked an existential pair and also did first and second named projections of a pair. Therefore we have to weaken it to make it fit the type requirements.

```
convertCont (‘go[ w' ] u) =
  ‘go-cc[ w' ] "" (convertValue (‘λ "y" ∶ ‘Unit ⇒ CPS.Terms.⊆-cont-lemma there u))
```

Now we are moving the term u, which is in the world w', into a lambda function, and then applying the lambda conversion process we defined above. This will allow us to hoist it in the next step, so that we can refer to it by its name. The empty string is planned as a placeholder for the name we will assign in lambda lifting in subsection 3.4. Since we are adding a new variable "y" to the environment, we have to weaken the expression u.

3.4. Lambda lifting. Our goal with closure conversion was that we would be able to move the lambda terms as we like. We want to move all of them to the

beginning so that we can give them names and refer to them with their specific names later. This will especially be necessary for the network communication: we need names to know which functions are sending and receiving data at a given point.

In our lambda lifting process, we do not define a new language like we did before. Since none of the terms need to change, we can keep using the closure language. Now we will define a function that changes the program structure.

```
entryPoint : ∀ {w}
  → [] ⊢ ⋆< w >
  → Σ (List (Id × Type × World))
      (λ newbindings → All (λ {(_ , σ , w') → [] ⊢ ↓ σ < w' >}) newbindings
      × (toCtx newbindings) ⊢ ⋆< w >)
```

The program structure in the previous language is just a closed continuation $[] \vdash \star< w >$. In the new structure, we will have a list of closed terms, which are actually the lambda terms we are relocating, and a continuation that should be able to use the list of closed terms. However we cannot simply have a list of terms, because the type of terms change according to the context, type and world. We need some notion of a list that can hold values of different types. Moreover, we do not want a loose definition, we want to use the same list to be the context of the remaining continuation. What we can do is to use an existential pair to say that there exists a list of triples of names, types and worlds that satisfy a certain property. If we call this property $P : \mathsf{List}\ (\mathsf{Id} \times \mathsf{Type} \times \mathsf{World}) \to \mathsf{Set}$, then the existential type would be written as $\Sigma\ (\mathsf{List}\ (\mathsf{Id} \times \mathsf{Type} \times \mathsf{World}))\ (\lambda\ xs \to P\ xs)$.[15]

[15] Σ in Agda is standard library is the type for existential pairs. It basically stands for "there exists". In constructive logic, to prove that $\exists x.P(x)$ holds, you would choose an x and then show

Now we need to specify what P is. Since we have two things we have to show, we can use a product, i.e. pair.

The first part of the product will be a special kind of list called All.[16] We will use it to guarantee that our list will be of the exact same length as the list in the existential quantifier, and to ensure that each term in the list is of the right type. The second part of the product is the remaning continuation, but we are adding the list of gathered terms to the context. Since we started the entire process with a closed term, we will not have anything else in the context.

The definition of entryPoint requires functions to lift values and continuations. We state their types as such:

```
liftValue : ∀ {Γ τ w} → ℕ
  → Γ ⊢₀ ↓ τ < w >
  → ℕ × Σ (List (Id × Type × World))
    (λ newbindings → All (λ {(_ , σ , w') → [] ⊢ ↓ σ < w' >}) newbindings
    × (Γ +++ toCtx newbindings) ⊢ ↓ τ < w >)
liftCont : ∀ {Γ w} → ℕ
  → Γ ⊢₀ ⋆< w >
  → ℕ × Σ (List (Id × Type × World))
```

that $P(x)$ holds. In Agda, we follow the same path. For the type Σ A (λ a → P a), we first write a term a : A and then write a term of the type P a to prove that it holds.

[16]The type All is defined in Agda standard library, in Data.List.All. Morally it has the same constructors as a list, but it takes a predicate of P : A → Set and a list xs : List A. Each element in the list is mapped to a Set using the predicate, and a value of type All P xs is constructed by providing values of type P x for each x in xs. Let's see a simple example: the type All (λ n → n ≡ n) (1 :: 2 :: 3 :: []) can take a value refl :: refl :: refl :: [], note that the first refl is of type 1 ≡ 1, and the second is of type 2 ≡ 2 and so on. Also note that the constructors [] and _::_ are overloaded and they can be used for both lists and the type All.

```
  (λ newbindings → All (λ {(_ , σ , w') → [] ⊢ ↓ σ < w' >}) newbindings
  × (Γ +++ toCtx newbindings) ⊢ ⋆< w >)
```

Types of these two functions resemble entryPoint, except they have a natural number in their arguments and outputs, and they account for the context of the initial term. The number in the term is a simple hack to generate new names for the lambda terms. It is increased by one for each lambda and accumulated throughout the entire lifting process.

The most interesting case in these two functions is how lambdas are handled.

```
liftValue {Γ} {_} {w} n (`λ x ∶ σ ⇒ t) with liftCont n t
... | n' , xs , Δ , t' =
  suc n'
  , ("_lam" ++ show n' , ` σ cont , w) :: xs
  , (`λ x ∶ σ ⇒ t) :: Δ
  , `v ("_lam" ++ show n') (++ʳ Γ (here refl))
```

Our return type is a product of four values. Let's go over each:

1. The accumulator for the name generator. In this case it is increased by one because we just used n' in naming the lambda.
2. The first part of the existential Σ (List (Id × Type$_o$ × World)) _. We now have one more lambda to gather, so we have to add one more term with the appropriate name, type and world.
3. The actual term itself to the All list we defined above.
4. What the lambda will be replaced with. Since the lambda now has a name and it is in the context of the resulting term, use the `v term to refer to it instead of having the literal lambda term.

In the closure language, we changed ‘go to ‘go-cc in a way that the term inside will always be a function. This guarantees that the network computation function will always be lifted and given a name. We want to use this name during our network communication, which is why we have a string argument in ‘go-cc. We can save that name during lambda lifting.

```
liftCont n (‘go-cc[ w' ] str u) with liftValue n u
... | n' , xs , Δ , u' = n' , xs , Δ , (‘go-cc[ w' ] ("_go" ++ show n) u')
```

To add a name tag to ‘go-cc, we are using the unupdated number n because it will be the value to be used in the top-most level lambda, which is in fact the function we want to access later.

The other cases of these two functions consist of recursive calls, list append equality proofs and calls to the weakening lemma. The actual code is verbose and not very interesting for our purposes.

3.5. Lifted monomorphic. Until this point, our definition of Hyp for every language contained two constructors: one for a specific world, and valid values. However there is no way to denote valid values in JavaScript, therefore we want to convert all valid values to specific world references. We are defining a new language that we will call LiftedMonomorphic. The idea is to add one copy in client and another one in server, for each valid value. Our new Hyp definition is as follows:

```
data Hyp : Set where
  _⁰_<_> : (x : Id) (τ : Type) (w : World) → Hyp
```

The LiftedMonomorphic language is in most cases the same as the previous language. The only exceptions are the two terms that contain valid values. Since we do not have valid values we have to revise them.

$$
\begin{aligned}
&\text{`put\_`=\_`in\_} : \forall\,\{\tau\ w\}\ \{m : \tau\ \mathsf{mobile}\} \to (u : \mathsf{Id}) \\
&\quad \to \Gamma \vdash \downarrow \tau < w > \\
&\quad \to ((u \wr \tau < \mathsf{client} >) :: (u \wr \tau < \mathsf{server} >) :: \Gamma) \vdash \star < w > \\
&\quad \to \Gamma \vdash \star < w >
\end{aligned}
$$

The first of the terms to revise is putting a mobile term in a valid value. Instead of adding a valid value of $u \sim (\lambda \to \tau)$ in the context, we put two values in different worlds.

$$
\begin{aligned}
&\text{`lets\_`=\_`in\_} : \forall\,\{C\ w\} \to (u : \mathsf{Id}) \\
&\quad \to \Gamma \vdash \downarrow (\text{`}\mathbin{⌘}\ C) < w > \\
&\quad \to ((u \wr C\ \mathsf{client} < \mathsf{client} >) :: (u \wr C\ \mathsf{server} < \mathsf{server} >) :: \Gamma) \vdash \star < w > \\
&\quad \to \Gamma \vdash \star < w >
\end{aligned}
$$

Similarly for shamrock, we add two different values to the context instead of a given valid value. However since C is a function $\mathsf{World} \to \mathsf{Type}$ we actually have to call it with the corresponding world.

Also for each primitive that is a valid value, such as logging, we define two primitives, one copy for client and another for server.

That concludes the definition of our monomorphic language. We also define and prove weakening lemmas for continuation expressions and values, with the names $\subseteq$-cont-lemma and $\subseteq$-term-lemma respectively.

3.5.1. *Monomorphization.* Even though defining the terms that are different in the LiftedMonomorphic language shows most of the conversion process, we still need to state the types of the conversion functions. Similar to the previous conversions,

we will have a function to convert continuations and values. We will state their types as follows:[17]

```
convertValue : ∀ {Γ τ w}
  → Γ ⊢ₒ ↓ τ < w >
  → (convertCtx Γ) ⊢ᵐ ↓ convertType τ < w >
convertCont : ∀ {Γ w}
  → Γ ⊢ₒ ⋆< w >
  → (convertCtx Γ) ⊢ᵐ ⋆< w >
```

The types of these functions are similar to the ones we had before. For a given value or continuation in the previous language, we get a value or continuation of the corresponding type and context. What makes this process a bit more interesting is that contexts are not in one to one correspondence anymore, because we replace every valid value with two things. This means convertCtx and proofs that show that a given variable is in the context will have to change.

```
convertCtx : Contextₒ → Contextᵐ
convertCtx [] = []
convertCtx ((x ∶ τ < w >) :: xs) = (id , convertType τ , w) :: convertCtx xs
convertCtx ((u ∼ C) :: xs) =
  (u ∶ convertType (C client) < client >) :: (u ∶ convertType (C server) < server >) :: convertCtx xs
```

Since a converted context now can be longer than a given context, we have to redefine convert∈, a function that takes a proof that a hypothesis is in the context in the previous language and returns a proof that the converted hypothesis is in the converted context in the new language.

[17] We will use the suffix $_o$ for the types of the closure language formalization, and ^{m} for the types of the LiftedMonomorphic formalization.

```
hypLocalize : Hyp_o → World → Hyp^m
hypLocalize (x ⦂ τ < w >) w' = x ⦂ convertType τ < w >
hypLocalize (u ∼ C) w = u ⦂ convertType (C w) < w >
convert∈ : ∀ {ω} → (Γ : Context_o) → (h : Hyp_o)
  → h ∈ Γ
  → hypLocalize h ω ∈ convertCtx Γ
convert∈ _ (x ⦂ τ < w >) (here refl) = here refl
convert∈ {client} _ (u ∼ C) (here refl) = here refl
convert∈ {server} _ (u ∼ C) (here refl) = there (here refl)
convert∈ {ω} ((x ⦂ τ < w >) :: xs) h (there i) = there (convert∈ {ω} xs h i)
convert∈ {ω} ((u ∼ C) :: xs) h (there i) = there (there (convert∈ {ω} xs h i))
```

Using these functions, it becomes trivial to define the ‘v and ‘vval cases of convertValue.

Now let's use all of our definitions and state the overall conversion function for our entire program. During lambda lifting, we gathered a list of closed terms, i.e. terms with an empty context. What we are left with is a term that calls certain terms from the list we gathered. Our goal with the overall function is to keep the same structure, but all the types and terms will be in the new language.

```
entryPoint : ∀ {w}
  → Σ (List (Id × Type_o × World))
    (λ newbindings → All (λ {(_ , σ , w') → [] ⊢_o ↓ σ < w' >}) newbindings
    × (toCtx_o newbindings) ⊢_o ⋆< w >)
  → Σ (List (Id × Type^m × World))
    (λ newbindings → All (λ {(_ , σ , w') → [] ⊢^m ↓ σ < w' >}) newbindings
      × (toCtx^m newbindings) ⊢^m ⋆< w >)
```

The actual function definition is trivial but verbose, it converts each term separately and then shows that the contexts are still equal.

4. Formalization of JavaScript

The last step of our compiler is code generation, and it is not practical to generate target code directly as a concatenation of strings. The common practice is to have an abstraction of the target language, and then generate the code using that abstraction.

However to ensure that we are compiling to a language that executes without exception, we will formalize a subset of JavaScript that enforces certain type and context restrictions. A simply typed JavaScript, if you will.

JavaScript is an imperative language that distinguishes between statements and expressions. We want to reflect the difference between these two. However, it is considered a good practice to avoid declaring or defining a variable in the global namespace. The most common way to do so in JavaScript is to define everything in an anonymous function (i.e. lambda) that is instantly called.

LISTING 1. An anonymous function that is instantly called.

```
(function() {
  // definitions etc.
})();
```

Since JavaScript has function scope, any declaration made with `var` will stay in the scope of the anonymous function and will not be accessible from outside.

However, notice that the code above is an expression statement, which is a kind of statement that evaluates an expression and ends. For example `3;` is a statement that evaluates a number expression. A function call is just another

kind of expression, which makes `console.log("hello");` another example of such statements.

Notice that this statement does not add any new declarations to the global namespace. We want to disallow declarations to the global namespace, so variable declarations should not be accessible if it is not in a function definition. Therefore, we will define two different types, namely statements and function statements.

Let's start by defining the types that will be used in our JavaScript formalization:

```
data Type : Set where
  ‘Undefined ‘Bool ‘Number ‘String : Type
  ‘Function : List Type → Type → Type
  ‘Object : List (Id × Type) → Type
  ‘Σt[t×[_×t]cont] : Type → Type
```

This definition tells us that our base types are ‘Undefined, ‘Bool, ‘Number and ‘String. The function type takes the list of types to denote the arguments and a return type. The object type takes a list of all key names and the types of data each key holds.[18] The last one is a hacky solution to the hard coded existential pair we saw in the previous languages. In reality it is a JavaScript object that does not obey the typing rules we want for objects, therefore we added a separate type to deal with it.

We have the notions of hypothesis and conclusion as we defined in the previous languages.

```
data Hyp : Set where
  _∘_<_> : (x : Id) (τ : Type) (w : World) → Hyp
```

[18]Remember that JavaScript objects are like Python dictionaries.

```
Context = List Hyp

data Conc : Set where
  _<_> : (τ : Type) (w : World) → Conc
```

Now we will define three types: statements, function statements and expressions. Since they are defined mutually recursively, let's go over their notation first in order to understand the big picture.

We will first define `Stm_<_> : Context → World → Set`, and `Stm Γ < w >` should read "the statement under the context Γ in the world w".

We will then define `FnStm_⇓_꞉_<_> : Context → Context → Maybe Type → World → Set`, and `FnStm Γ ⇓ γ ꞉ mσ < w >` should read "the function statement under the context Γ that extends the context with γ and has returned the function with type mσ, in the world w".

Our last definition will be `_⊢_ (Γ : Context) : Conc → Set`, and `Γ ⊢ τ < w >` should read "the expression under the context Γ, of the type τ, in the world w".

Now we can move on to the actual definitions of these notions.

4.1. Statements. We explained above that defining things in the global namespace is considered bad practice, and therefore we planned to place all of the code in an anonymous function that will be called instantly. Therefore, we only need one kind of statement that can hold the function call. By definition, we ruled out the possibility of changing the global namespace, therefore we are not accounting for the context after the statement.[19]

```
data Stm_<_> : Context → World → Set where
  'exp : ∀ {Γ τ w} → Γ ⊢ τ < w > → Stm Γ < w >
```

[19]Note that this does not mean purity. The function calls we can make can still have side effects.

4.2. Function statements. Now let's define rules for statements that are allowed in function definitions.

```
data FnStm_⇓_⨟_<_> : Context → Context → Maybe Type → World → Set where
  ‘nop : ∀ {Γ w mσ} → FnStm Γ ⇓ [] ⨟ mσ < w >
  ‘exp : ∀ {Γ τ w mσ} → Γ ⊢ τ < w > → FnStm Γ ⇓ [] ⨟ mσ < w >
```

We define a no operation statement for the cases that we do not want to output any real JavaScript statement. Obviously it does not change the context or affect the return value in any way.

Our next definition is the expression statement that we explained earlier. It does not change the local context or the return value in the function.

```
‘var : ∀ {Γ τ w mσ} → (id : Id)
  → (t : Γ ⊢ τ < w >)
  → FnStm Γ ⇓ (id ⨟ τ < w > :: []) ⨟ mσ < w >
```

Variable declaration is the first kind of statement that can change the context; it will add a single element to the context.

```
_;_ : ∀ {Γ γ γ' w mσ}
  → FnStm Γ ⇓ γ ⨟ mσ < w >
  → FnStm (γ ++ Γ) ⇓ γ' ⨟ mσ < w >
  → FnStm Γ ⇓ (γ' ++ γ) ⨟ mσ < w >
```

We are now defining what we can call a combination statement. Since we determined FnStm to be a single entity and not a list of statements, we need a statement that can hold two different statements.[20] The context changes in this definition is important in order to understand our formalization.

[20]Semicolon cannot be used in names in Agda, so we used the Unicode character U+FF1B, which looks almost the same.

Since we have a variable declaration FnStm, we know that a given FnStm can possibly change the context, which means we have to take γ into account. Then for the second statement, the initial context will be the resulting context of the first statement, which is γ ++ Γ. Overall, the FnStm resulting from the combination of two smaller ones will have the initial context Γ and will add γ' ++ γ to the local context.

```
_;return_ : ∀ {Γ γ τ w}
  → FnStm Γ ⇓ γ ⨟ nothing < w >
  → (γ ++ Γ) ⊢ τ < w >
  → FnStm Γ ⇓ γ ⨟ (just τ) < w >
```

We define a similar statement to return the function we are in. Note that no other function statement allows a change in the Maybe Type[21] argument of the FnStm. In the beginning, that argument will start as nothing, because the function has not yet been returned. It can only change to just τ through this return statement we are defining. Therefore having just τ guarantees that the function has a return statement. Moreover, the structure of this statement encourages us to use it only at the very end of the FnStm.[22]

Later we will define all functions in a way that the FnStm they take as an argument must have a return statement.

```
'if_'then_'else_ : ∀ {Γ γ γ' w mσ}
  → Γ ⊢ 'Bool < w >
  → FnStm Γ ⇓ γ ⨟ mσ < w >
```

[21]Remember that Maybe in Agda (and Haskell) is the same as `option` type in ML variants.

[22]It is possible to create a FnStm such as ('nop ;return 'undefined) ; 'exp 'undefined, but notice that the second half of the combination statement will not be executed in reality since the function will return beforehand.

```
  → FnStm Γ ⇓ γ' ⨟ mσ < w >
  → FnStm Γ ⇓ γ ∩ γ' ⨟ mσ < w >
```

JavaScript provides two different kinds of conditionals, ternary expression and imperative if/else blocks. We are choosing to use the latter, and to define ‘if_‘then_‘else_ as a FnStm that takes other function statements.

However two different branches in an if/else block may define different variables and hence end up with different resulting contexts. A possible solution to that is to force the two branches to have exactly the same resulting contexts, which would turn out to be impractical later in the conversion from our last language to JavaScript. A simpler option is to have two different contexts and get their intersection, which means even if two branches define different variables, we will later only have access to the ones defined in both.

```
‘prim : ∀ {Γ h mσ w} → (x : Prim h) → FnStm Γ ⇓ (h :: []) ⨟ mσ < w >
```

The last FnStm we will define is the primitive. Primitives work like they do in the previous languages, it adds a reference or definition of the primitive to the local context. The definition of Prim : Hyp → Set contains the corresponding terms for the previous primitives such as ‘alert, ‘readFile, ‘log etc. In addition, we have two more primitives, socket references for the client and server. We will go in detail about them in section 5.

4.3. Expressions. Now let’s define the notion of expressions, starting with the simple ones.

```
data _⊢_ (Γ : Context) : Conc → Set where
  ‘string : ∀ {w} → String → Γ ⊢ ‘String < w >
  ‘true : ∀ {w} → Γ ⊢ ‘Bool < w >
```

```
‘false : ∀ {w} → Γ ⊢ ‘Bool < w >
‘_∧_ : ∀ {w} → Γ ⊢ ‘Bool < w > → Γ ⊢ ‘Bool < w > → Γ ⊢ ‘Bool < w >
‘_∨_ : ∀ {w} → Γ ⊢ ‘Bool < w > → Γ ⊢ ‘Bool < w > → Γ ⊢ ‘Bool < w >
‘¬_ : ∀ {w} → Γ ⊢ ‘Bool < w > → Γ ⊢ ‘Bool < w >
‘_===_ : ∀ {τ w} {eq : EqType τ} → Γ ⊢ τ < w > → Γ ⊢ τ < w > → Γ ⊢ ‘Bool < w >
‘n_ : ∀ {w} → (ℤ ⊎ Float) → Γ ⊢ ‘Number < w >
‘_⩽_ : ∀ {w} → Γ ⊢ ‘Number < w > → Γ ⊢ ‘Number < w > → Γ ⊢ ‘Bool < w >
‘_+_ : ∀ {w} → Γ ⊢ ‘Number < w > → Γ ⊢ ‘Number < w > → Γ ⊢ ‘Number < w >
‘_*_ : ∀ {w} → Γ ⊢ ‘Number < w > → Γ ⊢ ‘Number < w > → Γ ⊢ ‘Number < w >
```

We have string, boolean and number literals, and simple logical, numerical, and comparison operations defined on them. Note that JavaScript does not differentiate between integers and floating numbers, so we only have one type ‘Number, whose literal term accepts both $\mathbb{Z}$ and Float. [23]

The equals operator _===_ is more interesting than the other ones. We want to allow only certain types of values to be compared for equality. The === operator in JavaScript compares identity rather than equality, which is problematic for functions and objects, so we will only use the base types. It suffices to say that EqType : Type → Set similar to the _mobile we defined in the previous languages, we will skip the definition and continue JavaScript expressions.

```
‘undefined : ∀ {w} → Γ ⊢ ‘Undefined < w >
```

In JavaScript, the value `undefined` has the type `undefined`. This creates a type inconsistency, because when we declare a new variable in JavaScript without initiating it with a value, its initial value and type are `undefined`. Then when we

[23] Remember that ⊎ is the discriminated union type in Agda standard library, just like `Either` in Haskell.

assign it a value, the type of the variable changes. This sort of behavior should not be allowed in our formalization.

However having an undefined type and value is still useful for other purposes. If a function is not supposed to return anything, we can use ‘Undefined as its return type, and return the term ‘undefined, defined above, for formality.

$$\text{`v} : \forall \{\tau\ w\} \to (x : \mathsf{Id}) \to (x \wr \tau < w >) \in \Gamma \to \Gamma \vdash \tau < w >$$

We define an expression to refer to variables that are in the context.

$$\begin{aligned}
&\text{`}\lambda_\Rightarrow_ : \forall \{\mathsf{argTypes}\ \tau\ w\ \Gamma'\} \to (\mathsf{ids} : \mathsf{List\ Id}) \\
&\quad \to \mathsf{FnStm}\ ((\mathsf{zipWith}\ (_\wr_ < w >)\ \mathsf{ids}\ \mathsf{argTypes})\ +\!\!+\ \Gamma) \Downarrow \Gamma' \wr (\mathsf{just}\ \tau) < w > \\
&\quad \to \Gamma \vdash \text{`}\mathsf{Function\ argTypes}\ \tau < w > \\
&\text{`}\_\cdot\_ : \forall \{\mathsf{argTypes}\ \tau\ w\} \to \Gamma \vdash (\text{`}\mathsf{Function\ argTypes}\ \tau) < w > \\
&\quad \to \mathsf{All}\ (\lambda\ \sigma \to \Gamma \vdash \sigma < w >)\ \mathsf{argTypes} \\
&\quad \to \Gamma \vdash \tau < w >
\end{aligned}$$

Even though JavaScript is an imperative language, it supports anonymous (lambda) functions. However anonymous functions take statements, not expressions.[24]

Our lambda expression first takes a list of variable names that will be added to the context with the corresponding types we get from ‘Function argTypes τ, which tells us that argTypes is the list of types of the arguments, and τ is the return type of the function. Notice that the statements in the lambda are allowed to change the local context inside, but the lambda must have a return statement inside that returns a value of type τ.

[24]It is worth noting that ECMAScript 6, an update to JavaScript, introduces "arrow functions" that can take expressions.

Function calls take a function expression and a list of expressions, which are the arguments to the function. The list is in a special type called All that makes sure that each expression is of the type of the corresponding argument.

$$
\begin{aligned}
&\text{`obj} \;:\; \forall\,\{\mathsf{w}\} \to (\mathsf{terms} \;:\; \mathsf{List}\;(\mathsf{Id} \times \Sigma\;\mathsf{Type}\;(\lambda\,\tau \to \Gamma \vdash \tau < \mathsf{w} >)))\\
&\quad \to \Gamma \vdash \text{`Object}\;(\mathsf{toTypePairs}\;\mathsf{terms}) < \mathsf{w} >\\
&\text{`proj} \;:\; \forall\,\{\mathsf{keys}\;\tau\;\mathsf{w}\} \to (\mathsf{o} \;:\; \Gamma \vdash \text{`Object}\;\mathsf{keys} < \mathsf{w} >)\\
&\quad \to (\mathsf{key} \;:\; \mathsf{Id}) \to (\mathsf{key}\;,\;\tau) \in \mathsf{keys}\\
&\quad \to \Gamma \vdash \tau < \mathsf{w} >
\end{aligned}
$$

Now we define object literals. The type restriction of objects are functions are similar in our formalization, however this time we will not use the type All, because we already have a definition of $\in$ for lists and we want to make use of that. We defined our JavaScript type constructor $\text{`}\mathsf{Object}$ to take $\mathsf{List}\;(\mathsf{Id} \times \mathsf{Type})$, i.e. pairs of key names and JavaScript types of the values they will hold.

The function $\mathsf{toTypePairs}$ maps triples $(\mathsf{id}\;,\;\tau\;,\;\mathsf{t})$ to $(\mathsf{id}\;,\;\tau)$. When it is applied to the first argument of the constructor $\text{`}\mathsf{obj}$, it should give us the pair list that is necessary for the resulting type of the JavaScript object we are creating.

The next definition we are making is projection on objects, i.e. property access. If we have an object expression and a key that is included in the type of the object, then we can access the key.

$$
\begin{aligned}
&\text{`pack}\Sigma \;:\; \forall\,\{\sigma\;\mathsf{w}\} \to (\tau \;:\; \mathsf{Type})\\
&\quad \to \Gamma \vdash \text{`Object}\;((\texttt{"type"}\;,\;\text{`String}) :: (\texttt{"fst"}\;,\;\tau) ::\\
&\qquad (\texttt{"snd"}\;,\;\text{`Function}\;(\text{`Object}\;((\texttt{"type"}\;,\;\text{`String}) ::\\
&\qquad\quad (\texttt{"fst"}\;,\;\sigma) :: (\texttt{"snd"}\;,\;\tau) :: []) :: []) \;\text{`Undefined}) :: []) < \mathsf{w} >\\
&\quad \to \Gamma \vdash \text{`}\Sigma\mathsf{t}[\mathsf{t}\times[\;\sigma\;\times\mathsf{t}]\mathsf{cont}] < \mathsf{w} >\\
&\text{`proj}_1\Sigma \;:\; \forall\,\{\tau\;\sigma\;\mathsf{w}\} \to \Gamma \vdash \text{`}\Sigma\mathsf{t}[\mathsf{t}\times[\;\sigma\;\times\mathsf{t}]\mathsf{cont}] < \mathsf{w} > \to \Gamma \vdash \tau < \mathsf{w} >\\
&\text{`proj}_2\Sigma \;:\; \forall\,\{\tau\;\sigma\;\mathsf{w}\} \to \Gamma \vdash \text{`}\Sigma\mathsf{t}[\mathsf{t}\times[\;\sigma\;\times\mathsf{t}]\mathsf{cont}] < \mathsf{w} >
\end{aligned}
$$

```
  → Γ ⊢ ‘Function (‘Object (("type" , ‘String)
      :: ("fst" , σ) :: ("snd" , τ) :: []) :: []) ‘Undefined < w >
```

We also define the JavaScript type ‘Σt[t×[σ ×t]cont] to be constructed using an object with two fields, the first constructed by the type in the existential quantifier, the second is a function that also uses the same type inside.

```
‘serialize : ∀ {τ w} {m : τ mobile} → Γ ⊢ τ < w > → Γ ⊢ ‘String < w >
‘deserialize : ∀ {τ w} {m : τ mobile} → Γ ⊢ ‘String < w > → Γ ⊢ τ < w >
```

We also need to have restricted serialization and deserialization for our JavaScript expressions. In reality, we will call `JSON.stringify` for ‘serialize and `JSON.parse` for ‘deserialize. Remember that we defined a notion of mobility in our previous languages. We will define a similar one for JavaScript terms.

```
data _mobile : Type → Set where
  ‘Bool^m : ‘Bool mobile
  ‘Number^m : ‘Number mobile
  ‘String^m : ‘String mobile
  ‘Object^m : ∀ {pairs} → All (λ {(_ , τ) → τ mobile}) pairs → (‘Object pairs) mobile
```

Mobility of ‘Bool, ‘Number and ‘String does not need explanation. Mobility of ‘Object, however, requires all values held in the object to be mobile as well, using the All type we defined earlier. The anomaly in here is the lack of ‘Undefined, and the reason is that `JSON.stringify` ignores the keys that hold the value `undefined`.[25]

[25]For example `JSON.stringify({"a": undefined})` returns `{}`.

We now finished the formalization of a subset of JavaScript. We also state and define weakening lemmas for all three types we defined above, their proofs are simple inductions.

```
⊆-stm-lemma : ∀ {Γ Γ' w} → Γ ⊆ Γ' → Stm Γ < w > → Stm Γ' < w >
⊆-fnstm-lemma : ∀ {Γ Γ' γ mσ w} → Γ ⊆ Γ'
  → FnStm Γ ⇓ γ ⨟ mσ < w >
  → FnStm Γ' ⇓ γ ⨟ mσ < w >
⊆-exp-lemma : ∀ {Γ Γ' τ w} → Γ ⊆ Γ' → Γ ⊢ τ < w > → Γ' ⊢ τ < w >
```

Before we move on, we also define a function to get a default filler value for any type we want. We will use this as a hacky solution for when there is a value for which we have to satisfy the type. Since we do not have a value `null` or `undefined` that fits any type[26], we need a way to get a filler value.

```
default : ∀ {Γ w} (τ : Type) → Γ ⊢ τ < w >
default ‘Undefined = ‘undefined
default ‘Bool = ‘false
default ‘Number = ‘n (inj₁ (+ 0))
default ‘String = ‘string ""
default (‘Function τs σ) =
  ‘λ (map (underscorePrefix ∘ Data.Nat.Show.show) (downFrom (length τs))) ⇒
    ((‘exp ‘undefined ;return default σ))
default {Γ} {w} (‘Object fields) =
    eq-replace (cong (λ l → Γ ⊢ (‘Object l) < w >) (pf fields)) (‘obj (f fields))
  where
    f : List (Id × Type) → List (Id × Σ Type (λ τ → Γ ⊢ (τ < w >)))
```

[26]Our ‘undefined value has the value ‘Undefined so it is not a good candidate. This is an intentional design decision.

```
    f [] = []
    f ((id , τ) :: xs) = (id , τ , default {Γ} {w} τ) :: f xs
    pf : (xs : _) → toTypePairs (f xs) ≡ xs
    pf [] = refl
    pf (x :: xs) = cong (λ l → x :: l) (pf xs)
default (‘Σt[t×[_×t]cont] τ) =
    ‘packΣ ‘Undefined
      (‘obj (("type" , _ , ‘string "pair") ::
        ("fst" , _ , ‘undefined) ::
        ("snd" , _ , (‘λ ["o"] ⇒ (‘nop ;return ‘undefined))) :: []))
```

The base types return a value that is obvious. The interesting cases here are functions, objects and the existential pair. Our functions need to take a list of argument names, so we generate variable names like `_1` and `_0` for a function that takes two variables. Since variable names in JavaScript cannot start with numbers, we have to append some character to the beginning, so we chose underscore. Objects require a list of field names and terms, so we generate that and then prove that the list we generated satisfies the type of the object. Finally, the hardcoded existential pair needs to know what type to use. For convenience, we chose ‘Undefined. The rest of the case is straightforward from there.

For actual code generation, we write the following functions that convert the formalization above to a code string.

```
stmSource : ∀ {Γ w} → Stm Γ < w > → String
fnStmSource : ∀ {Γ Γ' mσ w} → FnStm Γ ⇓ Γ' ⨾ mσ < w > → String
termSource : ∀ {Γ c} → Γ ⊢ c → String
```

Their definitions consist of string concatenations with recursive calls.

Before any JavaScript code is written to file, we want to wrap them with basic code. For the server side, this code imports certain packages and starts and runs the server. For the client side, this code wraps the JavaScript code in HTML tags in makes it valid HTML code. They are basically string concatenations and their types are given below:

```
serverWrapper : String → String
clientWrapper : String → String
```

5. Conversion to JavaScript

5.1. Types of the conversion functions. Now that we defined a very strict subset of JavaScript, we will generate JavaScript code from our last language.

Before our last step, the structure of a program looks like this: The lambda lifted monomorphic language gives us a list of closed lambda terms and a term standing by itself that refers to those lambda terms. What we want to generate from this initially is a JavaScript statement for the client and the server. If we remember statements from subsection 4.1, they can represent an entire program on their own, by holding an immediately called anonymous function that contains the program. The overall function that converts the program from the lifted monomorphic language to JavaScript is as follows:

```
entryPoint : ∀ {w}
  → Σ (List (Id × Type^m × World))
    (λ newbindings → All (λ {(_ , σ , w') → [] ⊢^m ↓ σ < w' >}) newbindings
      × (LiftedMonomorphic.Types.toCtx newbindings) ⊢^m ⋆< w >)
  → (Stm [] < client >) × (Stm [] < server >)
entryPoint (xs , all , t) with convertλs all
... | (Γ' , Δ') , (cliFnStmLifted , s) , (serFnStmLifted , s')
```

```
  with convertCont {toCtx xs} {Γ' +++ []} {Δ' +++ []}
    {s = ++ˡ ∘ s} {s' = ++ˡ ∘ s'} _ t
... | (δ , cliFnStm) , (φ , serFnStm) =
  `exp ((` `λ [] ⇒ (cliFnStmLifted ; cliFnStm ;return `undefined) · []))
  , `exp ((` `λ [] ⇒ (serFnStmLifted ; serFnStm ;return `undefined) · []))
```

We already explained the type of this function, but what about its definition? The pattern matching (xs , all , t) represents our program: xs is the list of triples that denote the lifted lambdas, all is the proof using the All data type that those triples correspond to actual lambda terms, and t is the remaining program.

We pass all to another function convertλs in order to convert the lifted terms to a single function statement that consist of the converted JavaScript function declared with the proper name. Then we call convertCont with the remaining continuation expression t, and this provides us everything we need for the overall conversion. We call the only constructor of statements, `exp and pass it an anonymous function as we described in section 4.

Before stating the types of convertλ and convertλs, we have to define the function only that we will often use in our conversion function types in order to limit the JavaScript contexts in our conversions to a single world.

```
onlyCliCtx : Context → Context
onlyCliCtx [] = []
onlyCliCtx ((x ⦂ τ < client >) :: xs) = (x ⦂ τ < client >) :: onlyCliCtx xs
onlyCliCtx ((x ⦂ τ < server >) :: xs) = onlyCliCtx xs

onlySerCtx : Context → Context
onlySerCtx [] = []
onlySerCtx ((x ⦂ τ < server >) :: xs) = (x ⦂ τ < server >) :: onlySerCtx xs
```

```
onlySerCtx ((x ∶ τ < client >) :: xs) = onlySerCtx xs

only : World → Context → Context
only client xs = onlyCliCtx xs
only server xs = onlySerCtx xs
```

Clearly onlyCliCtx removes any non-client variable from a given context, and onlySerCtx removes any non-server variable from a context.

Using these definitions, let's state convertλ and convertλs to understand entryPoint better.

```
convertλ : (id : Id) (τ : Type^m) (w : World) → [] ⊢^m ↓ τ < w >
  → FnStm [] ⇓ ((id ∶ convertType τ < w >) :: []) ∶ nothing < w >
    × Σ _ (λ Γ → FnStm [] ⇓ Γ ∶ nothing < client >)
    × Σ _ (λ Δ → FnStm [] ⇓ Δ ∶ nothing < server >)
convertλs : ∀ {xs} → All (λ {(_ , σ , w') → [] ⊢^m ↓ σ < w' >}) xs
  → Σ _ (λ {(Γ , Δ) → (FnStm [] ⇓ Γ ∶ nothing < client >
    × only client (convertCtx (toCtx xs)) ⊆ Γ)
    × (FnStm [] ⇓ Δ ∶ nothing < server >
      × only server (convertCtx (toCtx xs)) ⊆ Δ)})
```

We will not go over their definitions, but their types are the key to understand the overall structure of our programs. Since the lambda lifted functions are closed terms, we can say that convertλ takes a closed term [] ⊢^m ↓ τ < w > and returns a triple that consists of an assignment function statement for the lambda it just converted, and two pairs of possible extensions of the context with function statements. The reason we need them is network communication. In the client and the server, we might need to add new definitions or make primitive function call for network communication purposes, and we do not know what

might need to be added to the context for that so we use Σ pairs. If we only had FnStm [] ⇓ ((id ⨟ convertType τ < w >) :: []) ⨟ nothing < w > as the resulting JavaScript term, that would be very limiting for cases like that.

Notice that in practive, we only call convertλ on lambda terms, which have types that look like ‘ σ cont, however we generalize it with τ for convenience.

Similarly, we define a function convertλs, that takes an All list of those closed lambda terms, converts them all to function statements using convertλ, and then accumulates the resulting function statements as a single entity. Notice that the function statements are in a Σ pair, which means that we cannot guess what their resulting contexts will be. However, there is a property about their resulting contexts that we can prove. Since each lifted function in xs will be declared in the corresponding world, we can show that for all lifted functions, if a function is lifted, then there is an assignment in the same world in JavaScript to the same name with the converted type of the lifted function. In our types, this is shown as only client (convertCtx (toCtx xs)) ⊆ Γ and only server (convertCtx (toCtx xs)) ⊆ Δ).

Now that we dealt with the conversion of the lifted functions, we should define conversion functions for continuation expressions and values in the lifted monomorphic language. The upshot is that continuation expressions will convert to function statements and values will convert to JavaScript expressions. This is not a one-to-one correspondence, however thinking of the conversion this way helps us to locate things in the big picture.

```
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 >)
```

These types might look cryptic at the moment, so let's go step by step. Our first function convertCont converts a continuation expression into two pairs containing function statements and their resulting contexts. However, it also guarantees that when we convert Γ, the context of the initial continuation expression, and take the hypotheses in a specific world using only, that should be a subset of the context of the function statement in the same world.

convertValue's type is slightly more cryptic. We still have the same subset condition, but we also have a type for the value. The context of the monomorphic value, Γ is related to the context of the resulting value in the same way. Since we only want contexts that contain variables from a single world, we say that the context of the resulting value should be the converted context filtered in favor of the world we are in. This can be stated as only w (convertCtx Γ) in our formalization. A possible question is why we need function statements in the client and the server when we have a converted value. The answer to that question is that some values contain continuation expressions in them, such as lambda functions. When we compile lambda functions, we want to make sure that we can generate the necessary network communication code at the same time. One converted expression is not enough to handle all of that, ergo we need to be able to generate function statements as well.

Before we get into the specific of these functions, let's define convertType.

```
convertType : {w : World} → Type^m → Type_j
convertType ‘Int = ‘Number
convertType ‘Bool = ‘Bool
convertType ‘Unit = ‘Object (("type" , ‘String) :: [])
convertType ‘String = ‘String
convertType {w} ‘ τ cont = ‘Function [ convertType {w} τ ] ‘Undefined
convertType {w} (‘ τ × σ) =
  ‘Object (("type" , ‘String) :: ("fst" , convertType {w} τ)
    :: ("snd" , convertType {w} σ) :: [])
convertType {w} (‘ τ ⊎ σ) =
  ‘Object (("type" , ‘String) :: ("dir" , ‘String) ::
    ("inl" , convertType {w} τ) :: ("inr" , convertType {w} σ) :: [])
convertType {w} (‘Σt[t×[_×t]cont] τ) = ‘Σt[t×[_×t]cont] (convertType {w} τ)
convertType {w} (‘Env Γ) = ‘Object (hypsToPair w Γ)
```

The base types are converted directly. The case of ‘ τ cont is a function without a return value as we discussed in subsection 3.2. Therefore having ‘Undefined as the return type makes sense. Cases of ‘ τ × σ and ‘ τ ⊎ σ might look a bit cluttered, however we are merely making up some JavaScript objects that can represent these types. Especially the case of ‘ τ ⊎ σ is a hacky solution; we keep the fields for both possibilities in the sum type, and keep an additional field "dir" so see if the value is "in left", i.e. inl or "in right", i.e. inr. However, since an actual value of the type ‘ τ ⊎ σ will only provide one of those values, we have to fill the other field by making up a filler value. We can use the function default for this. The most interesting case in here is ‘Env. So far we only had a constructor ‘buildEnv that enclosed the context, and did nothing else. Now we are saying that

an environment will be a JavaScript object that holds mappings of names in the context to their values. The omission of the remaining types will be explained at the end of this section.

5.2. Conversion cases. Now that we have a better idea of the big picture, let's get into the specifics of this conversion. We will try to cover a few cases that encompass the general ideas about this conversion.

```
convertCont {Γ} {Δ} {Φ} {s = s} {s' = s'} client (‘if t ‘then u ‘else v)
  with convertValue {Γ} {Δ} {Φ} {s = s} {s' = s'} t
... | t' , (Δ' , tCli) , (Φ' , tSer)
  with convertCont {Γ} {Δ' +++ Δ} {Φ' +++ Φ}
    {s = ++ʳ Δ' ∘ s} {s' = ++ʳ Φ' ∘ s'} client u
... | (Δ'' , uCli) , (Φ'' , uSer)
  with convertCont {Γ} {Δ' +++ Δ} {Φ' +++ Φ}
    {s = ++ʳ Δ' ∘ s} {s' = ++ʳ Φ' ∘ s'} client v
... | (Δ''' , vCli) , (Φ''' , vSer) =
  (_ , (tCli ; (‘if ⊆-exp-lemma (++ʳ Δ' ∘ s) t' ‘then uCli ‘else vCli)))
  , (_ , (tSer ; (uSer ; ⊆-fnstm-lemma (++ʳ Φ'') vSer)))
```

The first case we want to handle is the conditional continuation expression. We have three terms that we have to convert. After the first conversion, we get Δ' and Φ', the additions to the environment after convertValue t. The next time we compile something, we use them in the context, as seen above in the calls convertCont client u and convertCont client v. Because of the context restrictions of the conditional function statement, i.e. ‘if_‘then_‘else, in our JavaScript formalization, we want both of those calls to have the same initial contexts. By

definition of the conditional function statement, the resulting different contexts are reconciled by taking their intersection.

Note that the code snippet above is only the client case of the conditional; since the conditional function statement has to be in a specific world, we often have to pattern match on the world and define the cases separately.

Also remember that for all terms, we generate the function statements for client and server, such as uCli and uSer. If the term u has any other term inside that does network communication, then it will need to have statements that deal with that, similar to the ones we will show later in this section. For that reason, we add the function statements from the recursive calls to our final result.

The server case of the conditional, and the case of ‘letcase x , y ‘= t ‘in u ‘or v are morally similar to the case above, therefore we can skip them.

```
convertCont {Γ} {Δ} {Φ} {s = s} {s' = s'} w (‘let x ‘=fst t ‘in u)
  with convertValue {Γ} {Δ} {Φ} {s = s} {s' = s'} t
convertCont {Γ} {Δ} {Φ} {s = s} {s' = s'} client (‘let x ‘=fst t ‘in u)
  | t' , (Δ' , tCli) , (Φ' , tSer)
  with convertCont {_} {_ :: (Δ' +++ Δ)} {Φ' +++ Φ}
    {s = sub-lemma (++ʳ Δ' ∘ s)} {s' = ++ʳ Φ' ∘ s'}
    client u
... | (Δ'' , uCli) , (Φ'' , uSer) =
  (_ , (tCli ; (‘var x (‘proj (⊆-exp-lemma (++ʳ Δ' ∘ s) t') "fst" (there (here refl))) ; uCli)))
  , (_ , (tSer ; uSer))
```

We will look at an ‘_×_ elimination case as an example of how we handle adding a variable to the context. Many continuation expressions are structured like ‘let..=..‘in.., therefore this will serve as a general case for them.

First we convert the monomorphic value t to a JavaScript value, which gives us Δ' and Φ', the additions to the environment after convertValue t, similar to the previous case. In the next step, we have to add one slot to the context of the world we want to define a variable in. This is also reflected in the subset proof s, which is an implicit argument to the convertCont call we make. We are using a lemma we defined before, which says that if we know two lists that have a subset relation, the relation is preserved if we add the same element to both. It is a simple induction proof.

```
sub-lemma : ∀ {l} {A : Set l} {Γ Δ : List A} {h : A} → Γ ⊆ Δ → (h :: Γ) ⊆ (h :: Δ)
```

Going back to the conversion case, once we have function statements and resulting contexts for the continuation expression, we can combine the function statements using _;_. However, we have to define the variable x, which in this case happens to be the first projection of a pair. Since we defined in convertType that pairs will turn into a certain kind of JavaScript objects, we do the projection on the JavaScript object.

Note that the snippet above does not include the server case, which is very similar, as it would be redundant.

Now we want to look at how we want to open an environment in the current context. The word open in here means to make the names in the environment accessible in the context.[27]

```
openEnv : ∀ {Γ δ Δ mσ w} → δ ⊆ Δ
  → Γ ⊢ⱼ (‘Object (hypsToPair w Δ) < w >)
  → FnStm Γ ⇓ only w (convertCtx δ) ⨟ mσ < w >
openEnv {δ = []} {w = client} s t = ‘nop
```

[27]The word open is borrowed from ML-variants, since they use the keyword `open` with modules.

```
openEnv {δ = []} {w = server} s t = `nop
openEnv {Γ} {δ = (x ⦂ τ < client >) :: δ} {Δ} {w = client} s t =
  openEnv {Γ} {δ} {Δ} (sub-lemma' s) t
  ; `var x (`proj (⊆-exp-lemma (++ʳ (onlyCliCtx (convertCtx δ))) t) x (hypsToPair∈ (s (here refl))))
openEnv {Γ} {δ = (x ⦂ τ < server >) :: δ} {Δ} {w = server} s t =
  openEnv {Γ} {δ} {Δ} (sub-lemma' s) t
  ; `var x (`proj (⊆-exp-lemma (++ʳ (onlySerCtx (convertCtx δ))) t) x (hypsToPair∈ (s (here refl))))
openEnv {Γ} {δ = (x ⦂ τ < server >) :: δ} {Δ} {w = client} s t =
  openEnv {Γ} {δ} {Δ} (sub-lemma' s) t
openEnv {Γ} {δ = (x ⦂ τ < client >) :: δ} {Δ} {w = server} s t =
  openEnv {Γ} {δ} {Δ} (sub-lemma' s) t
```

Observe that this is an induction on δ, which is a subset of Δ. This is a trick we had to do in order to simulate an induction on Δ. If we did an induction on Δ, then we would have to construct terms of type `Object (hypsToPair w Δ) < w > for diminishing Δ. The subset trick greatly facilitates the proof.

The goal of the openEnv function is to build a function statement out of variable declarations for each item in the environment object. If an item does not belong to the current world, then we skip it and make a recursive call. Otherwise, we declare a variable with a projection to the corresponding field in the environment object and make a recursive call.

We will not go over the `open t `in u case of convertCont since all it does is to call openEnv and then work out the proofs that the function statement generated here fits the subset restrictions we are required to satisfy.

Now we should look at the `buildEnv case of convertValue. The full definition of it consists of the object creation and proofs that the object we created satisfies the types. The noteworthy part of the case is where we build the object itself.

```
convertValue {Γ} {w = w} (‘buildEnv {Δ} pf) = _
  where
    envList : (Δ : Context^m) → Δ ⊆ Γ → (w : World)
      → List (Id × Σ Type_j (λ τ → only w (convertCtx Γ) ⊢_j (τ < w >)))
    envList [] s w = []
    envList ((x ⁘ τ < w' >) :: hs) s w with w decW w'
    ... | no q = envList hs (s ∘ there) w
    envList ((x ⁘ τ < w' >) :: hs) s .w' | yes refl =
      (x , convertType {w'} τ , ‘v x (convert∈ (s (here refl)))) :: envList hs (s ∘ there) w'
```

Note that we replaced the real definition with an underscore. However the important content here is the function envList. Let's remember from the definition of closures that ‘buildEnv takes a proof that $\Delta \subseteq \Gamma$ and returns $\Gamma \vdash \downarrow$ ‘Env $\Delta <$ w $>$. Therefore for the definition of envList, we can do an induction on Δ and adjust the subset proof at each step. If we are in the correct world, then we add a field that contains the name, type, and the value which is already in the context and is accessible through the subset proof. If we are in the wrong world, then we can discard the hypothesis we are looking at.

The other cases of convertValue are simple inductions in the style of convertCont cases we have seen. Here is an example for the pair introduction term conversion:

```
convertValue {Γ} {Δ} {Φ} {s = s} {s' = s'} (‘ t , u)
  with convertValue {Γ} {Δ} {Φ} {s = s} {s' = s'} t
... | (t' , (Δ' , tCli) , (Φ' , tSer))
  with convertValue {_} {Δ' +++ Δ} {Φ' +++ Φ} {s = ++^r Δ' ∘ s} {s' = ++^r Φ' ∘ s'} u
... | (u' , (Δ'' , uCli) , (Φ'' , uSer)) =
  (‘obj (("type" , _ , ‘string "and") :: ("fst" , _ , t') :: ("snd" , _ , u') :: []))
  , (_ , (tCli ; uCli)) , (_ , (tSer ; uSer))
```

The concepts that are at play here are already explained in the previous cases.

Going back to the definition of convertCont, we should define how function calls should be converted to JavaScript.

```
convertCont {Γ} {Δ} {Φ} {s = s} {s' = s'} w (‘call t u)
  with convertValue {Γ} {Δ} {Φ} {s = s} {s' = s'} t
... | (t' , (Δ' , tCli) , (Φ' , tSer))
  with convertValue {_} {Δ' +++ Δ} {Φ' +++ Φ}
    {s = ++ʳ Δ' ∘ s} {s' = ++ʳ Φ' ∘ s'} u
... | (u' , (Δ'' , uCli) , (Φ'' , uSer)) with w
... | client =
  (_ , (tCli ; (uCli ; ‘exp (⊆-exp-lemma (++ʳ Δ'' ∘ ++ʳ Δ' ∘ s) (‘ t' · (u' :: [])))))))
  , (_ , (tSer ; uSer))
... | server =
  (_ , (tCli ; uCli))
  , (_ , (tSer ; (uSer ; ‘exp (⊆-exp-lemma (++ʳ Φ'' ∘ ++ʳ Φ' ∘ s') (‘ t' · (u' :: [])))))))
```

A function call in the lifted monomorphic language consists of two values, so we start by converting each of them, similar to our previous cases. When we get the JavaScript expressions and function statements back, we will pattern match on the world, and start constructing a JavaScript function statement. We mentioned in subsection 5.1 that continuation statements and function statements, and values and JavaScript expressions will loosely match up. In that sense, even though a function call in JavaScript is an expression, it will match up a function statement here. Because of the CPS conversion, our functions are an action by themselves and they stand by themselves; they call their callback when the computation is finished. Therefore a function call should be converted into an expression statement ‘exp that contains the function call JavaScript expression. It

is also worth noting that our JavaScript functions take an All list as an argument, so we have to put the argument in a singleton All list in the function call.

As we stated in the introduction, the process of conversion to JavaScript is not entirely complete. We have not exactly specified the conversion rules for the modal types. We have a partially working idea for the ‘go-cc case. Let's take a closer look into ‘go-cc.

```
convertCont {Γ} {Δ} {Φ} {s = s} {s' = s'} client (‘go-cc[ client ] str t)
  with convertValue {Γ} {Δ} {Φ} {s = s} {s' = s'} t
... | t' , (Δ' , tCli) , (Φ' , tSer) =
  (_ , (tCli ; ‘exp (⊆-exp-lemma (++ʳ Δ' ∘ s) t'))) , (_ , tSer)
convertCont {Γ} {Δ} {Φ} {s = s} {s' = s'} server (‘go-cc[ server ] str t)
  with convertValue {Γ} {Δ} {Φ} {s = s} {s' = s'} t
... | t' , (Δ' , tCli) , (Φ' , tSer) =
  (_ , tCli) , (_ , (tSer ; ‘exp (⊆-exp-lemma (++ʳ Φ' ∘ s') t')))
```

These are the uninteresting cases of ‘go-cc. If we are already in the target world, we can just evaluate the expression here.

For the cases that do have to communicate, we have to add code that sends a signal to the other world and waits for the response. We are using the Socket.IO library[1] which is available for both front and back ends.

Let's see an example program to understand the logic here. The program is supposed to ask the client user a prompt, for a file name. Then it will send the user input to the server. The server will read the given file and send the file content back to the client. Then the client will print the content to the page. In our definition of ML5, this program can be written this way:

```
file : [] ⊢5 ‘Unit < client >
file =
  ‘prim ‘prompt ‘in
  ‘prim ‘readFile ‘in
  ‘prim ‘write ‘in
  (‘ ‘val (‘v "write" (here refl))
    · ‘get {m = ‘String^m}
      (‘ ‘val (‘v "readFile" (there (here refl)))
        · ‘get {m = ‘String^m} (‘ ‘val (‘v "prompt" (there (there (here refl))))
          · ‘val (‘string "Enter file name"))))
```

In the source language, this program would look like this:

```
prim prompt in (prim readFile in
  (prim write in (write (get (readFile (prompt "Enter file name"))))))
```

Now let's see what this program should look like in JavaScript.[28]

LISTING 2. Example program for Socket.IO

```
// The server side Node.js code
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);
var fs = require('fs');


app.get('/', function(req, res){
```

[28]Once again, this is not an output our compiler produces, it is hand-written code. The purpose of this code snippet is to show what we need in order to complete the network communication case of the conversion.

```
  res.sendfile('index.html');
});

io.on('connection', function(socket){
  console.log('a␣user␣connected');

  socket.on('disconnect', function(){
    console.log('user␣disconnected');
  });

  socket.on('file', function(filename){
    console.log("file␣req:␣" + filename);
    socket.emit("file", fs.readFileSync(filename).toString());
  });
});

http.listen(3000, function(){
  console.log('listening␣on␣*:3000');
});

////////////////////////////////
// The client side JavaScript code
////////////////////////////////
var socket = io();

socket.emit('file', prompt("Enter␣file␣name"));
```

```
socket.on('file', function(content){
  console.log(content);
  document.getElementById("result").innerHTML = content;
});
```

Going back to ‘go-cc, if we are in the client and trying to get data from the server, we we are supposed to add a call to `socket.on` in the client, and `socket.emit` from the server with the converted value. Also notice that in section 3.2.1, we converted ‘get to include two ‘go terms. Hence a network flow started by the client will work as follows: The client will emit to the server that it is making a request, and start listening for a response. The server will detect this, and start computation. When the computation is finished, the server will emit it back to the client. The client will detect this and continue the program. The name tags placed in ‘go-cc during lambda lifting will be used in this process.

This concludes the conversion to JavaScript. 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 : [] $\vdash_5$ ‘Unit < client > $\rightarrow$ String $\times$ String

compileToJS = (clientWrapper *** serverWrapper)

- $\circ$ (stmSource *** stmSource)
- $\circ$ LiftedMonomorphicToJS.entryPoint
- $\circ$ LiftedMonomorphize.entryPoint
- $\circ$ LambdaLifting.entryPoint
- $\circ$ CPStoClosure.convertCont
- $\circ$ ML5toCPS.convertExpr (λ v $\rightarrow$ CPS.Terms.‘halt)

6. Related work

We have studied the possible use of modal logic and its proof terms as a way to organize programs that require network communication between the client and the server in a web setting. As we mentioned throughtout the thesis, this is a spin-off of Murphy's dissertation[18], with the extension of verification in each step and using modern technologies like Node.js and Socket.IO. We extend his work by formalizing it in a different proof assistant, and formalizing the last step of conversion to JavaScript. Licata and Harper [15] has worked on an Agda formalization of ML5 before, which was inspiring for this thesis and it is cited in relevant spots. The description of modal logic judgments in Macedonio's dissertation[16], and the definition of the language λ_{rpc} in Jia and Walker's paper[13] were also interesting reads on the use of modal logic to represent distributed programs.

Related to the verification side of my thesis, Chlipala has worked on verified and certified compilers of functional languages. The certified type-preserving compiler described in [6] goes through similar compilation processes as we do in this thesis, using the Coq proof assistant.[5] Similarly his work in [8] explores the use of tactics in correctness proofs to overcome the amount of boilerplate code, which was a problem in this thesis as well. Long proofs of weakening lemmas of terms, type and hypothesis equality decidability could be avoided in a proof assistant that uses tactics.

During the design decisions of the JavaScript formalization in section 4, the work on F*[10] and λ_{JS}[11] was an inspiration. However these projects are attempting to embody the essential features of JavaScript, while it is enough to define a small simply typed subset for our purposes.

It is also worth mentioning the projects that do not attempt to use modal logic to express communication. Chlipala's work on the Ur/Web programming

language allows you to write front and back ends simultaneously and guarantees certain correctness properties about well typed programs.[7] Opa[3] is another example with a simpler type system and similar to this thesis, it compiles to JavaScript on both ends, using Node.js for the back end. It is also a superset of JavaScript, which allows usage of existing JavaScript libraries. Similarly, Eliom[24] is a superset of OCaml that allows writing a client-server application as a single program. Meteor[25] is another relevant project that integrates front end events with database and Socket.IO. There are also other functional web programming languages that do not handle communication but attempt to solve the infamous JavaScript problem such as Elm[9] and PureScript[29].

7. Conclusion

In this thesis we have defined a minimal modal logic based functional language called ML5 and implemented a compiler for it in Agda, and proved its static correctness. However, our language is so minimal that it is almost impossible to use it for a real task; it must be extended to include recursive functions, data type declarations, lists, records and maybe even type inference.[23] Without a doubt, this will make the verification of the language a much more dreadful task. One might even ask if we need to implement a practical compiler in a proof assistant, since it is unlikely that it will perform well. Despite its simplicity and the lack of the features listed above, our implementation of ML5 does not perform well. This is because at every step we have to call lemmas that run induction through the entire term, namely weakening lemma. This is because we do the conversion and the static correctness proof at the same time, intrinsically. A compiler that

[29] http://www.purescript.org/

runs through the steps using more efficient functions and proves static correctness extrinsically might be a better choice for a larger-scale practical compiler.[14]

By limiting Murphy's argument to solely client and server, we allowed ourselves to prove more properties about the conversion process itself. We attempted the formalization of the last step of conversion to JavaScript, which was not formalized by Murphy. Having a limited number of worlds was crucial in this step to specify that a the context of a JavaScript term in a specific world can only have variables in the same world. Even though we did not complete the compilation of `go-cc and the modal terms, our JavaScript formalization provides insight about how to prove the static correctness of those cases in the future.

The goal of this thesis was to formalize the JavaScript code generation and implement the entire compiler in the Agda proof assistant. Since we have non-trivial programs compiling and running properly, it is fair to say that we made a successful attempt to achieve both of these goals.

Bibliography

[1] Socket.io. http://socket.io/. [accessed 16-April-2017].

[2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge;New York;, 1992.

[3] Henri Binsztok, Adam Koprowski, and Ida Swarczewskaja. *Opa: Up and Running*. " O'Reilly Media, Inc.", 2013.

[4] Patrick Blackburn, Maarten d. Rijke, and Yde Venema. *Modal logic*, volume 53. Cambridge University Press, New York;Cambridge [England];, 2001.

[5] Pierre Castéran and Yves Bertot. Interactive theorem proving and program development. coq'art: The calculus of inductive constructions., 2004.

[6] Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *ACM Sigplan Notices*, volume 42, pages 54–65. ACM, 2007.

[7] Adam Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In *ACM Sigplan Notices*, volume 45, pages 122–133. ACM, 2010.

[8] Adam Chlipala. A verified compiler for an impure functional language. In *ACM Sigplan Notices*, volume 45, pages 93–106. ACM, 2010.

[9] Evan Czaplicki. Elm: Concurrent frp for functional guis. *Senior thesis, Harvard University*, 2012.

[10] Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. Fully abstract compilation to javascript.

ACM SIGPLAN Notices, 48(1):371–384, 2013.

[11] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of javascript. In *European conference on Object-oriented programming*, pages 126–150. Springer, 2010.

[12] HaskellWiki. The javascript problem. https://wiki.haskell.org/The_JavaScript_Problem. [accessed 16-April-2017].

[13] Limin Jia and David Walker. Modal proofs as distributed programs. In *European Symposium on Programming*, pages 219–233. Springer, 2004.

[14] Joomy Korkut, Maksim Trifunovski, and Daniel R Licata. Intrinsic verification of a regular expression matcher. 2016.

[15] Daniel R Licata and Robert Harper. A monadic formalization of ml5. *arXiv preprint arXiv:1009.2793*, 2010.

[16] Damiano Macedonio. *Logics for Distributed Resources*. PhD thesis, Ph. D. thesis in computer science TD-2006-2, Universita" Ca'Foscari" di Venezia (Jan 2006), 2006.

[17] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, May 1996.

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

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

[20] Frank Pfenning. Lecture notes on intuitionistic Kripke semantics, March 2008.

[21] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical. Structures in Comp. Sci.*, 11(4):511–540, August 2001.

[22] Frank Pfenning and Carsten Schürmann. System description: Twelf—a meta-logical framework for deductive systems. In *International Conference on Automated Deduction*, pages 202–206. Springer, 1999.

[23] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, 2002.

[24] Gabriel Radanne, Jérôme Vouillon, Vincent Balat, and Vasilis Papavasileiou. Eliom: tierless web programming from the ground up. 2016.

[25] Isaac Strack. *Getting Started with Meteor JavaScript Framework*. Packt Publishing Ltd, 2012.