MECHANIZING THE METATHEORY OF REWIRE

A Thesis presented to

the Faculty of the Graduate School

at the University of Missouri

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

---

by

THOMAS N. REYNOLDS

Dr. William L. Harrison, Dissertation Supervisor

DECEMBER 2019

The undersigned, appointed by the Dean of the Graduate School, have examined the dissertation entitled:

MECHANIZING THE METATHEORY OF REWIRE

presented by Thomas Reynolds, a candidate for the degree of Doctor of Philosophy and hereby certify that, in their opinion, it is worthy of acceptance.

Dr. William L. Harrison

Dr. Rohit Chadha

Dr. Khaza Anuarul Hoque

Dr. Gergely Bana

ACKNOWLEDGMENTS

Undoubtedly, I owe my advisor, William L. Harrison, a tremendous debt of gratitude. Without his intervention, I'd most likely be a logician living amongst the philosophers. It was a privilege to work in his lab and to learn from him over the years. His confidence in me completely changed the trajectory of not only my work, but of my life, as well.

I wish to thank Dr. Adam Procter, Dr. Ian Graves, and the rest of my colleagues from the Center for High Assurance Computing. In addition to being great sources of knowledge concerning ReWire, each of them provided advice and encouragement even long after they left. In particular, Adam Procter provided a great example and his presence in the early parts of my transition to the lab proved invaluable.

Of course, I'd like to thank the members of my dissertation committee: Professor Rohit Chadha, Professor Khaza Anuarul Hoque, and Professor Gergei Bana. Professor Chadha provided logical and practical insights that both amazed and inspired me over the years. Professor Hoque and his extensive knowledge of FPGAs challenged me to improve my own understanding of them. To this day, Professor Bana is still the only other person I've known that wanted to discuss the Fitting translation of S4-modal logic.

Lastly, I wish to thank my family. My mother Helen instilled in me a great sense of duty and a work ethic that fueled my passion to pursue what inspired me. My wife Jenna supported me through my studies and blessed me with a daughter that challenges us almost as much I challenged my mother. I am eternally grateful to them for everything they have done for me.

Contents

List of Tables

Table Page

List of Figures

Figure Page

ABSTRACT

The λ-calculus provides a simple, well-established framework for research in functional programming languages that readily lends itself to the use of *formal methods*—that is, the use of mathematically sound techniques and supporting tools—to describe and verify properties of programming languages, as well. This is no coincidence. After all, the λ-calculus formalizes the concept of *effective computability*, for all computable functions are definable in the untyped λ-calculus, making it expressively equivalent to recursive functions. In software, the expressiveness of functional languages is considered a strength. Functional approaches to language design, however, needn't be limited to software. In hardware, the expressiveness of functional languages becomes a major obstacle to successful hardware synthesis, for the reason that such languages are usually capable of expressing general recursion. The presence of general recursion makes it possible to generate expressions that run forever, never producing a well-defined value.

In this dissertation, we study two novel variants of the simply typed λ-calculus, representing fragments of functional hardware description languages. The first variant extends the type system, using natural numbers representing time. This addition, though simple, is non-trivial. We prove that this calculus possesses bounded variants of type-safety and strong normalization. That is to say, we show that all well-typed expressions evaluate to values within a bound determined by the natural number index of their corresponding types. The second variant is a computational λ-calculus that formalizes the core fragment of the hardware description language known as ReWire. We prove that the language has type-safety and is strongly normalizing—the proof of strong normalization is the first mechanized proof of its kind. We define an equational theory with respect to this language. This allows us to prove that the language has desirable security properties by construction. This work supports a full-fledged, formal methodology for producing high assurance hardware.

Chapter 1

Introduction

This dissertation investigates the formal verification of the metatheory of simply typed λ-calculi. The simply typed λ-calculus, invented by Alonzo Church [19, 18, 20], forms the backbone of modern functional programming languages [8], such as ML [66], OCaml [110] and Haskell [78]. By a *formal verification* of some target **X**, we mean a proof of correctness (or that some property is true) of **X** with respect to a formal specification of **X**. Because hand-written proofs have proven problematic, the reliability of any formal verification depends crucially on the methods and tools used in proofs.[1] This has led to the development of a myriad of tools used in formal methods including: *model checkers* such as SPIN [48], SAL [70], and SMV [61], *automated theorem provers* such as Prover9 [23] and Automath [24], and *interactive theorem provers* such as Agda [75], Isabelle [25] and Coq [26].[2]

This work examines extensions of simply typed λ-calculi formalized in Coq. ReWire is a subset of Haskell—from which circuits are synthesized automatically. The language, design and implementation of ReWire has been introduced in previous work [83, 43, 42, 47]. The ReWire Core Calculus (RWC) is a computational λ-calculus *à la* Moggi [69] that embodies the "barebones" of ReWire. We formalize RWC in Coq and prove that the language possesses some desirable metatheoretic properties such as *type-safety*—the property that

[1] For example, although the main theorem and result in [55] are correct, several lemmas are false and part of the proof of the main theorem is incorrect.

[2] This list is not intended to be exhaustive. It merely provides a sampling of some of the more popular tools used in formal methods.

all well-typed expressions are either values or they can be further evaluated to a well-typed expression—and *strongly normalizing*—the property that all well-typed expressions evaluate to values. We also define an equational theory with respect to this language that allows us to demonstrate that the language possesses certain security properties by construction [45].

We define another extension the simply typed λ-calculus that incorporates natural numbers as indices into the type system. The addition of these indices is non-trivial. We demonstrate this by proving that this calculus possesses bounded variants of type-safety and strong normalization. That is to say, we show that all well-typed expressions evaluate to values within a bound determined by the natural number index of their corresponding types.

1.1 Overview

Chapter 2 presents the bounded time calculus. We introduce the type system and proceed to discuss novelties thereof. This work has no comparable machine-checked formalizations in the literature.

Chapter 3 presents work previously published on the rewire core calculus. We discuss the implementation and design of this calculus. A number of formal and novel techniques are developed and discussed. This dissertation concludes with a discussion of future work in Chapter 4.

Chapter 2

The Bounded Time Calculus

This chapter presents a variant of the simply typed λ-calculus. The type system for this calculus has been augmented with natural numbers, intended to represent a coarse grained approximation of computation time. The formalization contains proofs of many standard properties of the simply typed λ-calculus such as type safety and strong normalization.

2.1 Introduction

Type based approaches to termination add size parameters to type system as a means to guarantee that recursive functions terminate. The typing rule ListFix illustrates a (simplified) type-based approach to using size variables in recursive definitions (adapted from [10, 90]):

$$\frac{\Gamma, f : [T]^{\boldsymbol{n}} \to U \vdash e : [T]^{\hat{\boldsymbol{n}}} \to U}{\Gamma \vdash \texttt{fix} f := e : [T]^{\infty} \to U} \text{ (ListFix)}$$

where $\hat{}$ is the successor function, $\boldsymbol{n}$ is a size variable and $[T]^{\boldsymbol{n}}$ denotes the type of lists (with elements of type T) of a size no greater than $\boldsymbol{n}$. This requires each instance of f to be defined on lists smaller than e, and hence, each recursive call reduces the size parameter.

In this chapter, we present a variant of the simply typed λ-calculus inspired by sized types. This calculus extends a standard Church-style type system of the simply typed λ-calculus in two ways.[1] First, in this system, function types are the only types themselves

[1]Here we focus on Church-style approaches to typing, as opposed to typing Curry-style [29, 30].

that have parameters in a manner similar to the ListFix example above. Other types, such as products, sums and unit remain unchanged. Second, type judgments incorporate a variable as part of the judgment.

Whereas other approaches focus on far more expressive systems such as the Calculus of Inductive Constructions, we focus on a limited extension of the simply typed λ-calculus. The trade off in expressiveness facilitates the use of standard machinery to prove interesting metatheoretic properties of our system. Indeed, our formalization uses a standard approach to term construction and has been fully verified in Coq.

2.2 Background

The concept of using types for termination dates as far back as [65]. The underlying motivation for using sized types is that it aids in termination checking, as subsequent calls may be type checked for reduced size. Hughes et al. [53] incorporate sized types into a functional language.[2] In the system introduced in [53] each name for a datatype, i.e., `List`, `Stream`, represents a collection of nat-indexed datatypes such as $\texttt{List}^n$ where n is a size bound. In this system, sizes are a linear function of size variables and typing rules reinforce a requirement that each input generates an output of a smaller size. This supports a basic check for responsiveness of program in a reactive system because programs that are well-typed in this system will satisfy a *liveness* property–that every input eventually produces an output.

Building on the system introduced in [53], Pareto [77] examines an extension of Haskell with sized types. This extension utilizes linear sized types–including addition and constants. It provides a type-checking algorithm, as well.

Interest in sized types is not limited to functional programming–they have been incorporated into dependent type theories, too. Giménez [37] considers an extension of the Calculus of Constructions [27]. Sizes are not explicitly represented but still present

[2]For more references beyond those listed here, see [1, 89].

nonetheless. Other type systems involve more complex size algebras. For example, a more expressive language using linear sized types was introduced in [90] by extending the Calculus of Inductive Constructions with (co-)inductive types and size annotations. Other systems introduce sizes as upper bounds [9, 2], or add sized types in a dependently typed framework with polymorphism and indexed types [111]. Each of these systems has more expressive power than our own.

Whereas other approaches examine far more expressive languages, we focus on a limited extension of the simply typed λ-calculus. The trade off in expressiveness facilitates the use of standard machinery to prove interesting metatheoretic properties of our system. Indeed, our formalization uses a standard approach to term construction and has been fully verified in Coq.

2.3 BTC: The Bounded Time Calculus

In this section, we present the syntax and semantics of the Bounded Time Calculus (BTC). Along the way, we present the Coq encoding and a more readable version of each concept. We adopt the following conventions. We use s, t, u to denote terms, v, w to denote values, $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ to denote arbitrary variables, and T, U for types.

2.3.1 Syntax

We begin with BTC types. These include standard types such as products ($T \times U$ or *TProd T U*), sums ($T + U$ or *TSum T U*), and unit ($()$ or *TUnit*). Most importantly, there is a variant of the standard function or arrow type ($T \xrightarrow{\boldsymbol{n}} U$ or *TArrow n T U*). This is stated in Definition 2.1 and encoded in Figure 2.1.

Definition 2.1 (Types)**.** The set Ty of BTC types is defined thusly:

$$T, U \in Ty ::= T \xrightarrow{\boldsymbol{n}} U \mid T \times U \mid T + U \mid ()$$

where $\boldsymbol{n}$ denotes an arbitrary natural number.

```
Inductive Ty : Type :=
  | TArrow : nat → Ty → Ty → Ty
  | TProd : Ty → Ty → Ty
  | TSum : Ty → Ty → Ty
  | TUnit : Ty.
```

Figure 2.1: Coq Syntax for BTC Types

The variable decorating the function arrow represents a restriction on the time it takes to convert an argument to its corresponding output. We return to the significance of this below.

Terms and values are given standard definitions. This is stated in Definition 2.2 and encoded in Figure 2.3.1.

Definition 2.2 (Terms)**.** The set $term$ of BTC terms is defined thusly:

$$s, t, u \in term ::= \boldsymbol{x} \mid app\ t\ u \mid \lambda\, \boldsymbol{x}\ T\ t \mid \texttt{nil} \mid pair\ t\ u$$
$$\mid \pi_1\, t \mid \pi_2\, t \mid inl\ t\ T \mid inr\ t\ T \mid case\ s\ t\ u$$

Definition 2.3 (Values)**.** The set $value$ of BTC values is given by the following:

$$v, w \in value ::= \lambda\, \boldsymbol{x}\ T\ t \mid \texttt{nil} \mid pair\ v\ w \mid inl\ v\ T \mid inr\ v\ T$$

```
Inductive term : Type :=
  | var : id → term
  | app : term → term → term
  | λ : id → Ty → term → term
  | nil : term
  | pair : term → term → term
  | π1 : term → term
  | π2 : term → term
  | inl : term → Ty → term
  | inr : term → Ty → term
  | case : term → term → term → term.
```

Figure 2.2: Coq Syntax for BTC Terms

```
Inductive value : term → Prop :=
  | v_abs : ∀ x T t,
      value (λ x T t)
  | v_unit : value nil
  | v_pair : ∀ v w,
      value v → value w → value (pair v w)
  | v_inl : ∀ v T,
      value v → value (inl v T)
  | v_inr : ∀ v T,
      value v → value (inr v T).
```

Figure 2.3: Coq Syntax for BTC Values

The terms and values are by and large standard. We comment only on the term constructor $case$. This term, used for destructing sum types, takes three subterms: the first is a term of type $T + U$, the second is a function t from T to S, and the third is a function u from U to S. If the first subterm evaluates to $inl\ v\ U$ (resp., $inr\ v\ T$), then v will be passed to t (resp., u).

Before moving on to discuss the type system, we define free variable substitution. As this suggests, we must first provide a clear statement of what it means for a variable to occur free in a term. This is provided by Definition 2.4.

Definition 2.4 (Free Variables)**.** For any term t, the set of free variables in t, $FV(t)$ is defined as:

$$
\begin{aligned}
\boldsymbol{FV}(x) &= \{x\} \\
\boldsymbol{FV}(\mathit{app}\, t\, u) &= \boldsymbol{FV}(t) \cup \boldsymbol{FV}(u) \\
\boldsymbol{FV}(\lambda\ x\ T\ t) &= \boldsymbol{FV}(t) \setminus \{x\} \\
\boldsymbol{FV}(\texttt{nil}) &= \{\} \\
\boldsymbol{FV}(\mathit{pair}\, t\, u) &= \boldsymbol{FV}(t) \cup \boldsymbol{FV}(u) \\
\boldsymbol{FV}(\pi_1\, t) &= \boldsymbol{FV}(t) \\
\boldsymbol{FV}(\pi_2\, t) &= \boldsymbol{FV}(t) \\
\boldsymbol{FV}(\mathit{inr}\, t\, U) &= \boldsymbol{FV}(t)
\end{aligned}
$$

$$\boldsymbol{FV}(\mathit{inl}\, u\, T) = \boldsymbol{FV}(u)$$
$$\boldsymbol{FV}(\mathit{case}\, s\, t\, u) = \boldsymbol{FV}(s) \cup \boldsymbol{FV}(t) \cup \boldsymbol{FV}(u)$$

When $\boldsymbol{FV}(t) = \emptyset$, then t is *closed*.

Definition 2.5 (Substitution)**.** We now define the substitution of v for free occurrences of x in t, written '$t[x := v]$', thusly:

$$\begin{aligned}
x[x := v] &= v \\
y[x := v] &= y \qquad\qquad \text{if } y \neq x \\
(\mathit{app}\, t\, u)[x := v] &= \mathit{app}\, (t[x := v])\, (u[x := v]) \\
(\lambda\, x\, T\, t)[x := v] &= \lambda\, x\, T\, t \\
(\lambda\, y\, T\, t)[x := v] &= \lambda\, y\, T\, (t[x := v]) \quad \text{if } y \neq x \text{ and } y \notin \boldsymbol{FV}(v) \\
\texttt{nil}[x := v] &= \texttt{nil} \\
(\mathit{pair}\, t\, u)[x := v] &= \mathit{pair}\, (t[x := v])\, (u[x := v]) \\
(\pi_1\, t)[x := v] &= \pi_1\, (t[x := v]) \\
(\pi_2\, t)[x := v] &= \pi_2\, (t[x := v]) \\
(\mathit{inr}\, t\, U)[x := v] &= \mathit{inr}\, (t[x := v])\, U \\
(\mathit{inl}\, u\, T)[x := v] &= \mathit{inl}\, (u[x := v])\, T \\
(\mathit{case}\, s\, t\, u)[x := v] &= \mathit{case}\, (s[x := v])\, (t[x := v])\, (u[x := v])
\end{aligned}$$

In addition to the clauses for substitution into lambda abstractions, one typically also finds the following:

$$(\lambda\, y\, T\, t)[x := v] = \lambda\, z\, T\, (t[y := z])[x := v] \quad \text{if } z \notin \boldsymbol{FV}(t) \textit{ and } z \notin \boldsymbol{FV}(v)$$

When the other clauses for abstraction apply, this clause generates α-equivalent

expressions—that is, expressions equivalent up to a renaming of bound variables. In our setting, adding such a clause would only complicate matters for the reason that the other clauses suffice.

2.3.2 Type System

Typing rules for terms are given in Definition 2.6. Typing judgments take the form

$$\Gamma \vdash t : T^{\boldsymbol{n}}$$

where $\Gamma = \{\boldsymbol{x}_1 : T_1, \ldots, \boldsymbol{x}_m : T_m\}$ such that for each assumption $\boldsymbol{x}_i : T_i$, $\boldsymbol{x}_i$ denotes a term variable unique to Γ and T_i is a type (as defined in Definition 2.1). The set Γ is commonly referred to as a *context* or *environment*. In cases where Γ is empty, we write $\{\}$. Additionally, we write $\Gamma, \boldsymbol{x} : T$ as shorthand for $\Gamma \cup \{\boldsymbol{x} : T\}$.

We say that the expression $\boldsymbol{n}$ *decorates* the type T in $T^{\boldsymbol{n}}$. The range of expressions allowed as decorators is determined by the following grammar:

$$n, m \in \mathbb{N} ::= n \mid n + m \mid \max(n, m)$$

Though restrictive, this linear structure suffices for our needs here. The expression that decorates function types is more restrictive—only allowing natural numbers as decorators.

Definition 2.6 (Typing Judgments)**.** We define typing judgments thusly

$$\frac{}{\Gamma, \boldsymbol{x} : T \vdash \boldsymbol{x} : T^0}\ (\textsc{Var}) \qquad \frac{\Gamma, \boldsymbol{x} : T \vdash t : U^n}{\Gamma \vdash \lambda\, \boldsymbol{x}\, T\, t : (T \xrightarrow{n} U)^0}\ (\textsc{Abs})$$

$$\frac{\Gamma \vdash f : (T \xrightarrow{n} U)^m \quad \Gamma \vdash t : T^p}{\Gamma \vdash app\ f\ t : U^{(n+m+p+1)}}\ (\textsc{App}) \qquad \frac{}{\Gamma \vdash \texttt{nil} : ()^0}\ (\textsc{Nil})$$

$$\frac{\Gamma \vdash t : T^n \qquad \Gamma \vdash u : U^m}{\Gamma \vdash pair\, t\, u : (T \times U)^{n+m}}\ (\textsc{Pair}) \qquad \frac{\Gamma \vdash t : (T \times U)^n}{\Gamma \vdash \pi_1\, t : T^{(n+1)}}\ (\textsc{Pi}_1) \qquad \frac{\Gamma \vdash t : (T \times U)^n}{\Gamma \vdash \pi_2\, t : U^{(n+1)}}\ (\textsc{Pi}_2)$$

$$\frac{\Gamma \vdash t : T^n}{\Gamma \vdash inl\, t\, U : (T + U)^n}\ (\textsc{Inl}) \qquad \frac{\Gamma \vdash u : U^n}{\Gamma \vdash inr\, u\, T : (T + U)^n}\ (\textsc{Inr})$$

$$\frac{\Gamma \vdash s : (T + U)^n \quad \Gamma \vdash t : (T \xrightarrow{l} S)^m \quad \Gamma \vdash u : (U \xrightarrow{r} S)^p}{\Gamma \vdash case\ s\ t\ u : S^{(n+\max(l+m, r+p)+2)}}\ (\textsc{Case})$$

Figure 2.4 contains the Coq code defining the typing relation for BTC.

```
Inductive has_type : context → term → Ty → nat → Prop :=
  | Var : ∀ Γ x T,
      Γ x = Some T →
      Γ ⊢ var x : T // 0
  | Abs : ∀ Γ x T U t n,
      extend Γ x T ⊢ t : U // n →
      Γ ⊢ λ x T t : T -n-> U // 0
  | App : ∀ T U Γ f T n m p,
      Γ ⊢ f : T -n-> U // m →
      Γ ⊢ t : T // p →
      Γ ⊢ app f t : U // (n + m + p + 1)
  | Nil : ∀ Γ,
      Γ ⊢ nil : () // 0
  | Pair : ∀ Γ T U t u n m,
      Γ ⊢ t : T // n →
      Γ ⊢ u : U // m →
      Γ ⊢ pair t u : T × U // (n + m)
  | Pi1 : ∀ Γ T U t n,
      Γ ⊢ t : T × U // n →
      Γ ⊢ π1 t : T // (n + 1)
  | Pi2 : ∀ Γ T U t n,
      Γ ⊢ t : T × U // n →
      Γ ⊢ π2 t : U // (n + 1)
  | Inl : ∀ Γ T U t n,
      Γ ⊢ t : T // n →
      Γ ⊢ inl t U : T + U // n
  | Inr : ∀ Γ T U t n,
      Γ ⊢ t : U // n →
      Γ ⊢ inr t T : T + U // n
  | Case : ∀ Γ S T U s t u l r n m p,
      Γ ⊢ s : T + U // n →
      Γ ⊢ t : T -l-> S // m →
      Γ ⊢ u : U -r-> S // p →
      Γ ⊢ case s t u : S // (n + (max (l + m) (r + p)) + 2)
where "Γ ⊢ t : T // n" := (has_type Γ t T n).
```

Figure 2.4: BTC Typing Relation

The types for variables, abstractions and `nil` each have 0 as a decorator. Pairs inherit the sum of the decorators for the types of their subterms, while each type for the projection constructors adds one to the decorator of their subterm types. Sums possess the same decorators as the types of their subterms. For the application rule, the natural number decorating the arrow represents the time it takes to reduce a term of type T to a term of type U. In addition to natural number indexes, function types also receive an outer decoration. This represents the time for processing the function of that type. In line

with the other term constructors, the decorator for the resulting term adds 1. The rule for $case$ takes the $\max$ value of the decorators adorning either branch of the evaluation. Because this requires an additional term, it adds 2 to the decorator for the type of the case expression.

The type system possesses a property common to many simply typed λ-calculi. This property is that every well-typed term has a unique type, as stated in Theorem 2.7.

Theorem 2.7 (Type Uniqueness). *If* $\Gamma \vdash t : T^n$ *and* $\Gamma \vdash t : U^m$*, then* $T = U$.

In this typing system, with the addition of decorators, this property was not guaranteed. Interestingly, the type system also possesses similar property for decorators, as stated in Theorem 2.8.

Theorem 2.8 (Decorator Uniqueness). *If* $\Gamma \vdash t : T^n$ *and* $\Gamma \vdash t : U^m$*, then* $n = m$.

This property enforces a uniformity of decorator assignments, so to speak. In this setting, such a property represents a good guarantee that the system has (at some) desirable properties. Additionally, we also have it that terms well-typed in the empty context, are well-typed in any context:

Theorem 2.9. *If* $\{\} \vdash t : T^n$*, then* $\Gamma \vdash t : T^n$.

Theorem 2.9 provides further reassurances that the addition of decorators does not drastically alter the traditional properties of the simply typed λ-calculus's type system.

Our type system and definition of values (from Definition 2.3) provide us with *canonical forms*—that is, a property of closed, well-typed values. Many proofs of metatheoretic properties tend to be organized around canonical forms. This greatly reduces the cases one needs to consider. Our canonical forms are the following.

Lemma 2.10. *If* $\{\} \vdash v : (T \xrightarrow{n} U)^m$ *and* v *is a value, then there exists* $x\, u$*, such that* $v = \lambda\, x\ T\ u$.

Lemma 2.11. *If* $\{\} \vdash v : (T \times U)^m$ *and* v *is a value, then there exists* $t\, u$*, such that* $v = pair\ t\ u$*.*

Lemma 2.12. *If* $\{\} \vdash v : (T+U)^m$ *and* v *is a value, then there exists* w *such that* $v = inl\ w\ U$ *or* $v = inr\ w\ T$*.*

Lemma 2.13. *If* $\{\} \vdash v : ()^0$*, then* $v = \texttt{nil}$*.*

Substitution (given in Definition 2.5 above) preserves typing judgments. This requires that if free variables occur in well-typed terms, then there must be a typing assignment for those variables relative to the context. As stated in Lemma 2.14.

Lemma 2.14. *If* $x \in \mathbf{FV}(t)$ *and* $\Gamma \vdash t : T^n$*, then there exists a* U *such that* $\{x : U\} \in \Gamma$*.*

From this Corollary 2.15 follows—namely, that a term is closed if it is well-typed in the empty context.

Corollary 2.15. *If* $\{\} \vdash t : T^n$*, then* t *is closed.*

Moreover, we have Lemma 2.16 as a consequence—that the context of a typing judgment does not alter typing judgments, so long as all each context maintains assignments of types to any free variables.

Lemma 2.16. *If* $\Gamma \vdash t : T^n$ *and, if, for all* x*,* $x \in \mathbf{FV}(t)$*,* Γ *and* Γ' *assign the same type to* x*, then* $\Gamma' \vdash t : T^n$*.*

Finally, we have Theorem 2.17—that is, the substitution operation preserves typing judgments when the term being substituted is a value.

Theorem 2.17. *If* $\Gamma, x : U \vdash t : T^n$*, value* v*, and* $\{\} \vdash v : U^m$*, then* $\Gamma \vdash (t[x := v]) : T^n$*.*

This is a more restricted version than what one typically sees. In most simply typed λ-calculi, no additional restriction is placed on terms being substituted into expressions. Our version adds the restrict that a value must be substituted. In theory, all that one needs is to restrict the decorator of the type for such terms as in Corollary 2.18.

Corollary 2.18. *If* $\Gamma, x : U \vdash t : T^n$ *and* $\{\} \vdash v : U^0$*, then* $\Gamma \vdash (t[x := v]) : T^n$.

In practice, no proofs hinge on which version one picks. The reason for this is simple. In BTC, all well-typed values have 0 as their decorator.

Theorem 2.19. *If* $\{\} \vdash t : T^n$ *and value* v*, then* $n = 0$.

2.3.3 Small-Step Operational Semantics

In this section, we describe a semantics for BTC in a *small-step operational semantics*–or, *structural operational semantics*—first introduced in [81].[3] As the name suggests, a small-step operational semantics defines computations for the terms of a language as single execution steps. This makes it ideal for our setting.

In our semantics, we use $\rightsquigarrow$ for the single-step reduction relation. The step relation is defined inductively using the rules stated in Definition 2.20. Figure 2.5 contains the Coq code defining the single step reduction relation for BTC.

Definition 2.20 (Step Relation)**.**

$$\frac{value\ v}{app\ (\lambda\, \boldsymbol{x}\ T\ t)\ v \rightsquigarrow [\boldsymbol{x} := v]t}\ \text{(ST\_AppAbs)}$$

$$\frac{t \rightsquigarrow t'}{app\ t\ u \rightsquigarrow app\ t'\ u}\ \text{(ST\_App1)} \qquad \frac{value\ v \quad u \rightsquigarrow u'}{app\ v\ u \rightsquigarrow app\ v\ u'}\ \text{(ST\_App2)}$$

$$\frac{t \rightsquigarrow t'}{pair\ t\ u \rightsquigarrow pair\ t'\ u}\ \text{(ST\_Pair1)} \qquad \frac{value\ v \quad u \rightsquigarrow u'}{pair\ v\ u \rightsquigarrow pair\ v\ u'}\ \text{(ST\_Pair2)}$$

$$\frac{value\ v \quad value\ w}{\pi_1\ (pair\ v\ w) \rightsquigarrow v}\ \text{(ST\_Pi1)} \qquad \frac{value\ v \quad value\ w}{\pi_2\ (pair\ v\ w) \rightsquigarrow w}\ \text{(ST\_Pi2)}$$

$$\frac{t \rightsquigarrow t'}{\pi_1\ t \rightsquigarrow \pi_1\ t'}\ \text{(ST\_Pi1E)} \qquad \frac{t \rightsquigarrow t'}{\pi_2\ t \rightsquigarrow \pi_2\ t'}\ \text{(ST\_Pi2E)}$$

$$\frac{t \rightsquigarrow t'}{inl\ t\ T \rightsquigarrow inl\ t'\ T}\ \text{(ST\_Inl)} \qquad \frac{t \rightsquigarrow t'}{inr\ t\ T \rightsquigarrow inr\ t'\ T}\ \text{(ST\_Inr)}$$

$$\frac{s \rightsquigarrow s'}{case\ s\ t\ u \rightsquigarrow case\ s'\ t\ u}\ \text{(ST\_Case)}$$

$$\frac{value\ v}{case\ (inl\ v\ \ T)\ t\ u \rightsquigarrow app\ t\ v}\ \text{(ST\_CaseL)} \qquad \frac{value\ v}{case\ (inr\ v\ T)\ t\ u \rightsquigarrow app\ u\ v}\ \text{(ST\_CaseR)}$$

[3] The semantics we present here is inspired by [79, 80]. However, our *case* expressions align more closely with a functional approach to programming languages such as Haskell.

```
Inductive step : term → term → Prop :=
  | ST_AppAbs : ∀ x T t v,
        value v →
        (app (λ x T t) v) ⇝ [x:=v]t
  | ST_App1 : ∀ t1 t1' t2,
        t1 ⇝ t1' →
        app t1 t2 ⇝ app t1' t2
  | ST_App2 : ∀ v t2 t2',
        value v →
        t2 ⇝ t2' →
        app v t2 ⇝ app v t2'
  | ST_Pair1 : ∀ t1 t1' t2,
        t1 ⇝ t1' →
        pair t1 t2 ⇝ pair t1' t2
  | ST_Pair2 : ∀ v t2 t2',
        value v →
        t2 ⇝ t2' →
        pair v t2 ⇝ pair v t2'
  | ST_Pi1 : ∀ v1 v2,
        value v1 →
        value v2 →
        π1 (pair v1 v2) ⇝ v1
  | ST_Pi2 : ∀ v1 v2,
        value v1 →
        value v2 →
        π2 (pair v1 v2) ⇝ v2
  | ST_Pi1E : ∀ t t',
        t ⇝ t' →
        π1 t ⇝ π1 t'
  | ST_Pi2E : ∀ t t',
        t ⇝ t' →
        π2 t ⇝ π2 t'
  | ST_InL : ∀ t1 T t1',
        t1 ⇝ t1' →
        inl t1 T ⇝ inl t1' T
  | ST_InR : ∀ t1 T t1',
        t1 ⇝ t1' →
        inr t1 T ⇝ inr t1' T
  | ST_Case : ∀ t1 t1' t2 t3,
        t1 ⇝ t1' →
        case t1 t2 t3 ⇝ case t1' t2 t3
  | ST_CaseL : ∀ v1 T t2 t3,
        value v1 →
        case (inl v1 T) t2 t3 ⇝ app t2 v1
  | ST_CaseR : ∀ v1 T t2 t3,
        value v1 →
        case (inr v1 T) t2 t3 ⇝ app t3 v1
where "t1 ⇝ t2" := (step t1 t2).
```

Figure 2.5: BTC Step Relation

The step relation has a useful property that is immediately provable.

Theorem 2.21. *If* $s \rightsquigarrow t$ *and* $s \rightsquigarrow u$, *then* $t = u$.

As stated in Theorem 2.21, this property is that the BTC step relation is deterministic. We discuss more properties of our semantics in the next section.

2.4 Metatheory

In this section we discuss the metatheoretic properties of BTC. In particular, *type safety* (Section 2.4.1) and *strong normalization* (Section 2.4.2) are covered. In standard approaches to proving metatheoretic properties of simply typed λ-calculi, it is common to define an additional step relation. This is typically the reflexive-transitive closure of the single step relation. Because the reflexive-transitive closure provides no information on the number of steps taken, we do not take this approach. Our interests demand something different. In our setting, we use $\overset{n}{\rightsquigarrow}$ to denote a natural number indexed extension of our step relation, stated in Definition 2.22.

Definition 2.22 (Nat Indexed Step Relation)**.**

$$\frac{}{t \overset{0}{\rightsquigarrow} t}\ (\textsc{Refl}) \qquad \frac{s \rightsquigarrow t \quad t \overset{\boldsymbol{n}}{\rightsquigarrow} u}{s \overset{\boldsymbol{n+1}}{\rightsquigarrow} u}\ (\textsc{Step})$$

This relation has many useful properties. The most important of which are stated in Lemma 2.23 and Theorem 2.24.

Lemma 2.23. *For s, t, u and i, j, we have the following properties of the indexed step relation:*

($\textsc{Inclusion}$) *If $t \rightsquigarrow u$, then $t \overset{1}{\rightsquigarrow} u$,*

($\textsc{Transitivity}$) *If $s \overset{i}{\rightsquigarrow} t$ and $t \overset{j}{\rightsquigarrow} u$, then $s \overset{i+j}{\rightsquigarrow} u$.*

The first property is an inclusion property—it tells us that the indexed relation includes $\rightsquigarrow$. The second property is a transitivity property—it tells us that indexed relation is transitive and that the indices are additive. Each of these properties and Definition 2.22 is used to prove Theorem 2.24.

Theorem 2.24 (Congruence)**.** *For each rule stated in Definition* 2.20*, there exists a corresponding version with $\rightsquigarrow$ replaced by $\overset{\boldsymbol{n}}{\rightsquigarrow}$. For rules* ST_AppAbs, ST_Pi$_1$, ST_Pi$_2$, ST_CaseL*, and* ST_CaseR*, $\rightsquigarrow$ is replaced by $\overset{1}{\rightsquigarrow}$.*

2.4.1 Type Safety

In small-step operational semantics, type safety is the combination of two properties: *progress* and *preservation*. Traditionally speaking, the former is the property that all well-typed terms are either values or they step to some other term. In our setting, we incorporate decorators into the mix. In the case of progress (Theorem 2.25), decorators play no additional role.

Theorem 2.25 (Progress)**.** *If $\{\} \vdash t : T^n$, then either t is a value or there exists u such that $t \rightsquigarrow u$.*

The same cannot be said of preservation (Theorem 2.26).

Theorem 2.26 (Preservation)**.** *If* $\{\} \vdash t : T^n$ *and* $t \rightsquigarrow u$*, then there exists* m *such that* $m < n$ *and* $\{\} \vdash u : T^m$*.*

When well-typed terms step, the decorator for the type of the term stepped-to must be strictly smaller than that of the decorator for the term stepped-from. That is, preservation guarantees a reduction in decorators.

When we replace the $\rightsquigarrow$ with its indexed counterpart, we gain a variant of preservation (stated in Corollary 2.27) that relates decorators to the natural number indexes for the indexed step relation.

Corollary 2.27. *If* $\{\} \vdash t : T^n$ *and* $t \overset{m}{\rightsquigarrow} u$*, then there exists* l *such that* $l \leq n - m$ *and* $\{\} \vdash u : T^l$*.*

This tells us that as a term reduces, the resulting decorator for its type has an upper bound determined by its initial decorator minus the number of steps taken.

Progress and preservation guarantee that well-typed terms never "get stuck," so to speak. That is to say, take any term t if t cannot step (by some application of rules from Definition 2.20), and t is not a value, then something has gone wrong in the process of computing $t - t$ is *stuck* in a stage where nothing can be done with it. Theorem 3.9 and Theorem 3.8 provide us with a guarantee that this situation will not happen with well-typed terms.

Corollary 2.28 (Soundness)**.** *If* $\{\} \vdash t : T^n$ *and* $t \rightsquigarrow u$*, then* u *is either a value or there exists* v *such that* $u \rightsquigarrow v$*.*

2.4.2 Strong Normalization

Normalization is a property of the step relation—often stated in terms of possible sequences of steps in the reduction of terms. A step (or reduction) relation is *weakly normalizing* if there exists a finite sequence steps ending in a *normal-form*—an irreducible term.

If every such sequence ends in a normal-form, we say that the step relation is *strongly normalizing*. In BTC, all values are normal-forms, so we use "value" in place of "normal-form" without any issues.[4]

Strong normalization has a special significance in hardware applications. The reason for this is simple. Functions realized in hardware cannot be allowed to "loop forever" between clock ticks. There must be a static, finite upper bound on the computation time between clock ticks.[5] Though not every λ-calculus is strongly normalizing, BTC is.[6] We show this by establishing that all well-typed BTC terms *terminate* (in the sense stated in Definition 2.29).

Definition 2.29 (Termination)**.** For any term t, t *terminates* iff there exists v, n such that $t \overset{n}{\rightsquigarrow} v$ and v is a value.

Our step and indexed step relations preserve termination (as stated in Lemma 2.30).

Lemma 2.30. *For all terms t, u,*

1. *If $t \rightsquigarrow u$, then t terminates iff u terminates.*
2. *If $t \overset{n}{\rightsquigarrow} u$, then t terminates iff u terminates.*

Definition 2.31 (Reducibility Sets)**.** For any term t, such that $\{\} \vdash t : T^n$ and t terminates, $t \in \boldsymbol{R}_T$ is determined by T:

$$
\begin{array}{lrll}
(T \text{ is } U \overset{m}{\to} V) & t \in \boldsymbol{R}_{(U \overset{m}{\to} V)} & \textit{iff} & \forall w, \text{ if } w \in \boldsymbol{R}_U, \text{then } (\mathit{app}\, t\, w) \in \boldsymbol{R}_V \\
(T \text{ is } U \times V) & t \in \boldsymbol{R}_{(U \times V)} & \textit{iff} & \exists m\, w, \mathit{value}\ w, t \overset{m}{\rightsquigarrow} w, \pi_1(w) \in \boldsymbol{R}_U \ \&\ \pi_2(w) \in \boldsymbol{R}_V \\
(T \text{ is } U + V) & t \in \boldsymbol{R}_{(U+V)} & \textit{iff} & \exists m\, w, \mathit{value}\ w, t \overset{m}{\rightsquigarrow} \mathit{inl}\ w\ U \ \&\ w \in \boldsymbol{R}_V \vee t \overset{m}{\rightsquigarrow} \mathit{inr}\ w\ V \ \&\ w \in \boldsymbol{R}_U \\
(T \text{ is } ()) & t \in \boldsymbol{R}_{()} & \textit{iff} & \exists m\, w, \mathit{value}\ w \ \&\ t \overset{m}{\rightsquigarrow} w
\end{array}
$$

The final clause for unit types is included only for completeness. Because only `nil` has unit as its type, and `nil` is a value, `nil` terminates since we have `nil` $\overset{0}{\rightsquigarrow}$ `nil`. In fact, for

[4] We discuss these topics again in Chapter 3, § 3.5.3.
[5] This issue is discussed in detail in [82, 83].
[6] Our proof follows methods introduced in [38, 99].

BTC if we had base, or atomic types, we would add the following clause to Definition 2.31:

$$(T \text{ is atomic}) \quad t \in \boldsymbol{R}_T \textit{ iff } \quad t \text{ terminates}$$

This clause for atomic types and the clause for unit types are equivalent.

We have some facts about reducibility sets—$\boldsymbol{R}$ sets for short—that follow from Definition 2.31.

Lemma 2.32. *For all terms t, u arbitrary n, m, and T,*

1. *If $t \in \boldsymbol{R}_T$, then t terminates,*
2. *If $t \in \boldsymbol{R}_T$, then there exists l such that $\{\} \vdash t : T^l$,*
3. *If $t \rightsquigarrow u$ and $t \in \boldsymbol{R}_T$, then $u \in \boldsymbol{R}_T$,*
4. *If $t \overset{n}{\rightsquigarrow} u$ and $t \in \boldsymbol{R}_T$, then $u \in \boldsymbol{R}_T$,*
5. *If $\{\} \vdash t : T^m$, $t \rightsquigarrow u$ and $u \in \boldsymbol{R}_T$, then $t \in \boldsymbol{R}_T$,*
6. *If $\{\} \vdash t : T^m$, $t \overset{n}{\rightsquigarrow} u$ and $u \in \boldsymbol{R}_T$, then $t \in \boldsymbol{R}_T$.*

In [38], the properties enumerated in Lemma 2.32 are labeled as conditions on reducibility sets—named 'CR' properties. Ours differ slightly, but remain close in spirit.

From Lemma 2.33—the $\boldsymbol{R}$-Substitution Lemma— it follows that the BTC is strongly normalizing. This lemma is more commonly referred to as the "Substitution Lemma."

Lemma 2.33 ($\boldsymbol{R}$-Substitution)**.** *Let $v_1, \ldots, v_n$ be values such that for each $i = \{1, \ldots, n\}$, $v_i \in \boldsymbol{R}_{V_i}$. If $\{x_1 : V_1, \ldots, x_n : V_n\} \vdash t : T^j$, then $(t[x_1 := v_1] \ldots [x_n := v_n]) \in \boldsymbol{R}_T$.*

By property 2 of Lemma 2.32, the assumption in Lemma 2.33 entails that for each v_i there exists an l such that

$$\{\} \vdash v_i : V_i^l$$

because for each v_i, we have $v_i \in \boldsymbol{R}_{V_i}$ (by assumption). The $\boldsymbol{R}$-Substitution property (from Lemma 2.33) entails the Strong Normalization Theorem (stated in Theorem 2.34) by using the empty context for the typing judgment.

Theorem 2.34 (Strong Normalization)**.** *If* $\{\} \vdash t : T^n$*, then* t *terminates.*

2.5 Conclusions

The interesting insight, now, comes from the combination of Corollary 2.27 and Theorem 2.34. To see this, first recall that Theorem 2.19 tells us that all values well-typed in the empty context have 0 as a decorator. This gives us a more specific version of our bounded version of preservation (Corollary 2.27), stated in Corollary 2.35.

Corollary 2.35. *If* $\{\} \vdash t : T^n$*,* $t \overset{m}{\rightsquigarrow} u$ *and value* u*, then* $0 \leq n - m$ *and* $\{\} \vdash u : T^0$*.*

This tells us something special about termination in our Strong Normalization Theorem. To see why, note that if $\{\} \vdash t : T^n$, then there exists v, m such that $t \overset{m}{\rightsquigarrow} v$ and v is a value. Corollary 2.35 tells us exactly how to find a good choice for m because m must be less than or equal to n.

Chapter 3

The ReWire Core Calculus

This chapter is from a conference paper [85] and a published paper [86] on the mechanization of a subset of ReWire's core language.

3.1 Abstract

Constructing high assurance, secure hardware remains a challenge, because to do so relies on both a verifiable means of hardware description and implementation. However, production hardware description languages (HDL) lack the formal underpinnings required by formal methods in security. Still, there is no such thing as high assurance systems without high assurance hardware. We present a core calculus of secure hardware description with its formal semantics, security type system and mechanization in Coq. This calculus is the core of the functional HDL, ReWire, shown in previous work to have useful applications in reconfigurable computing. This work supports a full-fledged, formal methodology for producing high assurance hardware.

3.2 Introduction

It is generally recognized that reconfigurable technology has a "programmability" problem [7, 3] and high-level synthesis (HLS) from functional languages is a commonly proposed remedy for this problem [33, 95, 13, 5, 14, 6, 34, 113]. Pure functional languages—i.e., those without side effects—support equational reasoning as a basis for program verifica-

tion. Combining the two—i.e., HLS from a pure functional language—provides a methodology for high assurance hardware as demonstrated in previous work by the authors [83, 43, 42, 47]. The current article addresses the formalization of this methodology by mechanizing the semantics for a pure HLS language—namely, ReWire—in the Coq theorem proving system [26], with the goal of combining the programmability advantages of functional hardware description with formalized reasoning. All of the definitions and theorems in this paper have been checked with the Coq proof checker; the Coq v8.5 scripts are downloadable [64].

ReWire is a functional hardware description language (HDL): it is a functional language—a subset of Haskell—from which circuits are synthesized automatically. Previous work has introduced ReWire's language design and implementation as well as its application to the construction of high assurance hardware [83, 43, 42, 47]. This article describes the Coq formalization of ReWire intended to support the verification of hardware designs and, in particular, the information flow properties described in our previous work [45, 43, 83].

The ReWire development flow is intended to approach that of functional programming to the greatest extent possible. First, device specifications are "roughed out" in Haskell, allowing testing and debugging in a familiar mode (e.g., using QuickCheck [21] as we did in Graves et al. [43]). Formal specification and verification typically starts at this point in the process. Refactoring into ReWire generally involves choosing base types (i.e., replacing Haskell's `Data.Word` with ReWire's built-in types). The ReWire compiler produces VHDL and vendor tools are used to synthesize, etc., to an FPGA. Making a formal methodology out of this requires a mechanized semantics for ReWire as a foundation for verification of designs and of the ReWire compiler. This article provides that foundation.

The aforementioned previous work, in panoramic view, used "by-construction" properties of layered monads to verify properties by hand. For the moment, we rely on the reader's intuition to explain the contributions of the present work at a high level (Sec-

tion 3.3 presents an overview of these concepts in more detail). Assume, for example, that ReWire devices h and l are written respectively in terms of state monad layers, `StT Hi` and `StT Lo`. Then, device h (resp., l) only accesses internal storage of type `Hi` (resp., `Lo`). In a composite device written in terms of monad `M = StT Hi (StT Lo Id)`, it is guaranteed by semantic properties of the layers `StT Hi` and `StT Lo` to disallow covert channels between the `Hi` and `Lo` storage.

The challenge is, then, the formalization of ReWire and, in particular, ReWire's underlying layered monad language and its semantic properties within an automated proof system. The contributions of this work are as follows. **(1)** A static effect-type system (extending and mechanizing Wadler's "marriage" of effects and monads [109]) that disallows covert storage channels in ReWire. This type system extends state layers with effect labels, so that, continuing the example above, h (resp., l) is written in monad `StT RW Hi (StT ⟨⟩ Lo Id)` (resp., `StT ⟨⟩ Hi (StT RW Lo Id)`). The effect label "`RW`" means h can both read and write on the `Hi` layer and while "$\langle\rangle$" means it can do neither on the `Lo` layer (and, *vice versa*, for l). The soundness of our type system (Theorems 3.19 and 3.21) guarantees freedom from covert storage channels. **(2)** A small-step semantics for ReWire formalized in Coq that justifies **(3)** a typed equational logic (Figure 3.14) capturing the semantic properties of monads and state layers used in by-hand proofs in our previous work. Finally, **(4)** a number of related metatheorems (e.g., progress, preservation, strong normalization, etc.) have been proved in Coq.

The direct approach to formalizing ReWire in Coq would be the transliteration of monad transformer declarations from Haskell into Coq, but this quickly runs afoul of Coq's strict positivity requirement. ReWire relies on reactive resumption monad transformers (see Section 3.3) for synchronous parallelism and this transformer is a coinductive construction, which can be tricky to formalize, even with Coq's coinduction library. Another approach considers formalizing ReWire's denotational semantics [82], building on existing work by Huffman [49] or Schröder and Mossakowski [94] in Isabelle/HOLCF. In-

stead, we chose to formalize a small-step, operational semantics for ReWire in Coq, in part, because the authors have more experience with Coq than with HOLCF, but also because developing and formalizing a small-step operational semantics seemed more straightforward than mechanizing denotational semantics. The semantic properties of ReWire's underlying monads on which the by-hand verifications of our previous work rely are then captured as an typed equational logic whose rules are derived from the formalized operational semantics.

The remainder of this section discusses related work. Section 3.3 presents an overview of ReWire to motivate the formal calculus, RWC. Section 3.4 defines the syntax and small-step operational semantics of RWC. Section 3.5 describes RWC's metatheory and a number of related metatheorems (e.g., progress, preservation, strong normalization, etc.) are demonstrated. A type-directed equational logic for RWC is defined in Section 3.6. Section **??** discusses conclusions and future work.

Related Work

Andrews [3] argues that a paradigm shift for reconfigurable computing is a necessary precondition for wider adoption of reconfigurable technology. Rather than focusing exclusively on performance metrics, the new paradigm must focus as well on what, for lack of a better term, might be called software engineering virtues—abstraction, modularity, program comprehensibility, productivity, rapid modifiability, reuse, and scalability, etc. What is required are programming models/languages for reconfigurable computing that embrace the software engineering virtues.

One proposed remedy to the programmability issue is high-level synthesis from functional languages [33], because, as originally observed by Sheeran [95], combinational logic has a functional flavor. More to the point, functional languages support the software engineering virtues through higher-order abstractions and type systems. ReWire provides the usual functional programming model of combinational logic—i.e., pure functions—but it

also provides a formal model of synchronous logic in the form of the reactive resumption monad discussed in Section 3.3.

There are a number of efforts to apply ideas and techniques from functional programming to hardware design and synthesis. Chisel [6] is a Scala-embedded domain-specific language developed as an implementation language for the RISC-V open source instruction set architecture[1]. Within the Haskell community, perhaps the most well known system for hardware synthesis is Lava [13]. Lava is a domain-specific language for hardware specification embedded in Haskell. Primitives in Lava are essentially structural and specify circuits at the level of signals. ReWire, by contrast, compiles a subset of Haskell itself to hardware circuits, and relies on an abstract set of behavioral primitives. The primary motivation for developing ReWire is as a vehicle for the design, implementation, and formal verification of high assurance hardware. There are some constructs of VHDL that have not been implemented in ReWire (e.g., tri-state buffers, multiple clock domains, etc.). We believe such constructs can be readily modeled in ReWire, but they have not been necessary for previous case studies [84, 83, 43, 42, 47, 46].

ForSyDe (Formal System Design)[2] is a formal design methodology that targets heterogeneous embedded systems [92, 91]. The ForSyDe toolset includes a system modeling language implemented as an embedded domain-specific language in Haskell that contains elements similar to those in ReWire, albeit not in resumption-monadic form. The ForSyDe methodology is based on refinement: high-level models are transformed semi-automatically into heterogeneous (i.e., mixed hardware and software) embedded systems. The ReWire methodology differs from that of ForSyDe in a number of respects. The ReWire language has type constructors for devices (described below in Section 3.3) that are compiled automatically into VHDL by the ReWire compiler, so hardware is generated directly from ReWire source code rather than produced by semi-automatic refinement. ForSyde targets heterogeneous hardware and software systems whereas ReWire

[1] https://riscv.org.
[2] https://forsyde.ict.kth.se/trac.

focuses on hardware exclusively. Finally, the formal methodology supported by ReWire, illustrated in previous publications [43, 84, 46, 83], is precisely that of pure functional languages; this is sometimes referred to as "Bird-Wadler" style program derivation (so-named after an influential textbook [12]). A Bird-Wadler derivation starts from a reference specification for an algorithm in a functional language and, through a series of semantics-preserving program transformations, produces a more efficient implementation. Desired properties of the implementation (e.g., correctness, security, etc.) are specified equationally and verified in terms of the reference semantics. The current work represents the formalization of the ReWire methodology in Coq.

Zhai et al. [113] consider high-level synthesis from recursive functional languages. Similarly, Cλash [5], is a compiler for a subset of Haskell to VHDL. Like ReWire, Cλash uses Haskell itself as a source language. Cλash requires some limits be placed on the kinds of algebraic data types used as well as the basic operating types. Both differ fundamentally from ReWire in that they require that a stack be constructed in hardware as part of the circuits they produce. It was an early design decision in the ReWire project to limit recursive functions to co-recursion (tail recursion) so as to obviate the need for a run-time stack or other unbounded data structures. Given hardware's fixed memory footprint, it seemed more natural to us to not require support for potentially unbounded data. Great care was taken in the design of ReWire so that it possesses a rigorous denotational semantics to support formal verification while maintaining synthesizability for all of its programs [82].

The Delite DSL compiler framework [56] seeks to address the "three P's" with respect to implementing software on parallel, heterogeneous systems. Delite addresses portability (i.e., retargetability of DSL compilers to a broad range of parallel hardware) through *language virtualization*. ReWire is also a virtualized DSL in that it has a separate compiler backend for producing FPGA-based implementations while reusing large parts of its host language's infrastructure—including Haskell's type system, front end, etc. In George, et

al., [34], the Delite framework is adapted to the generation of hardware from DSLs, specifically the hardware acceleration of kernels in a heterogeneous setting.

There is a vast literature on hardware security from an architectural or physical perspective, considering issues ranging from side channel attacks, hardware trojan detection, and the like. For an overview of this literature, please consult the references [51, 100, 105]. The architectural perspective of hardware security considers hardware structures or designs supporting security policies. To take one example from among many, GLIFT [101] is a gate-level information flow tracking method that inserts special "shadow circuits" to dynamically monitor all information flows within a circuit. The references include other examples of this architectural perspective [96, 52, 50, 11, 112, 104, 102, 103]. There is an orthogonal line of research in hardware security that considers the design, implementation and formal verification of hardware from a languages-based approach which we overview below; ReWire fits squarely within this research thrust.

Formal methods for secure hardware are generally spread across two categories: (1) type-based approaches [58, 59, 114]; and (2) logic-based approaches (including theorem-proving [63], and BDDs and model-checking [16]), in which a hardware design and desired properties are formulated in a logic and scrutinized in a (semi-)automatic manner. Types-based approaches have support for security concerns integrated into a domain-specific language for hardware description. With any security type system, the question of its expressiveness arises—i.e., does it reject secure designs? The types-based approach offers no recourse to the rejection of a secure design—you simply cannot argue with a type checker. A logic-based approach avoids this pitfall, but comes with overhead—e.g., your own theory of security—and neither is it connected directly to any implementation path.

Bluespec [74, 14] refers to a language and associated tools for hardware system design, specification, synthesis, modeling, and verification. There have been a number of incarnations of the Bluespec language since its inception in 2000, the first of which was as a Haskell subset extended with domain-specific operations for hardware design. The

Bluespec language seems to have evolved into its current form which is BSV (Bluespec SystemVerilog), which is no longer a functional language. There have been some formal methods tools developed for BSV [76, 87].

Braibant and Chlipala [15] apply ideas from CompCert [57] to hardware synthesis and is the most closely related to our own. Their work presents a certified compiler translating a monadic-functional HDL (called "Fe-Si") into RTL. Fe-Si is a small, idealized core of the BSV hardware description language [14]. Fe-Si's syntax is based on state monads, albeit not structured with monad transformers like ReWire's. Timing in Fe-Si is explicit, rather in the manner of VHDL, using an explicit clock tick parameter, whereas ReWire makes use of reactive resumptions as a basis for timing (see Section 3.3.2 below). One of the primary motivations behind the current work is to build a foundation for a verified compilation process for ReWire. Choi et al. [17] follow Braibant and Chlipala's work, starting from an idealized, BlueSpec-like language.

One language-based approach to hardware security is to extend an existing HDL with security types. Caisson [58], Sapper [59], and SecVerilog [114] each extend a subset of Verilog with security types and annotations. The type systems of Caisson and SecVerilog reject programs that violate information flow policies, while Sapper uses static analysis to automatically insert dynamic checks to enforce information flow policies at runtime. SecVerilog has an operational semantics, albeit not one formalized in a theorem prover with a proof system [71]. ReWire (or, RWC, rather) differs fundamentally from these language- and type-based approaches in three respects: (1) it is a pure functional language; (2) it possesses a formal semantics mechanized in Coq; and (3) its type system is based on effect types. We discuss the significance of item (3) in Section **??**.

The SAFE project focuses on the clean slate design of a provably secure computer system stack (e.g., hardware, operating system, etc.). In a recent publication [4], the SAFE team describes an operational semantics of the SAFE hardware's instruction set and its role in the end-to-end verification in Coq of a non-interference security property. The

ReWire project has complimentary, but orthogonal, goals to SAFE: developing a verifiable toolchain for producing high assurance, secure hardware. Interesting follow-on research would explore implementations of the SAFE hardware in the ReWire language.

One traditional approach to hardware verification starts from a design expressed in a production HDL, creates an abstract specification "by hand" as it were, encodes this specification in the logic of an automated theorem prover, and proceeds towards formal verification [63]. This approach relies heavily on the faithfulness of the abstraction step. One reason that this approach must be accomplished "by hand" is that production HDLs do not possess rigorous semantics. Although attempts have been made in the past to define them semantically, none of these projects were evidently completed [41, 54]. By contrast with production HDLs like Verilog or VHDL, ReWire possesses a rigorous semantics for which the present work provides a Coq mechanization. ReWire becomes a vehicle for expressing and implementing hardware designs and for verifying them as well. In previous work [43, 83], we presented several case studies in hardware verification based in ReWire, but there the verifications were not machine-checked.

Goncharov and Schröder [40] extend Moggi's computational λ-calculus with constructs for concurrency and shared state; RWC's design is inspired, in part, by their treatment of corecursion. Crary et al. [28] consider a logical characterization of information flow security that incorporates Moggi's computational λ-calculus at its core. With their approach, monads are, in effect, logical modalities signifying the potential presence of effects at a security level. In contrast, Harrison and Hook's treatment of information flow security [45] is more semantic and model-theoretic than Crary's logical and type-theoretic approach, relying on structural properties of monads and monad transformers to construct secure systems. Security verifications of ReWire designs [83] are based on Harrison and Hook's approach, and the present work formally supports that approach in Coq.

Ghica and Jung [35] provide a categorical semantics for a class of digital circuits in

terms of monoidal categories and are motivated by the need for supporting syntactic, equational reasoning. ReWire specifications may be reasoned about equationally in the usual manner of functional languages; this was the approach taken in our previous ReWire verification work [43, 83]. By contrast with Ghica and Jung's work, ReWire specifications are, more or less, ordinary functional programs that are compiled into circuits. Another categorical presentation of digital circuits is found in Megacz [62], who uses generalized arrows as a basis for hardware description.

Effect systems are a static semantics of effects while monads [69] are a dynamic semantics of effects. Effect systems [73] were initially associated with impure, strongly-typed functional languages in which the effect annotations make explicit the side effects already present implicitly in the language itself. Monads are used to mimic side-effecting computations within pure, strongly-typed functional languages (e.g., Haskell) in which there are no implicit side effects.

Layered monads—i.e., monads constructed by monad transformers [60]—provide modularity to the semantics of computational effects and functional programs alike by integrating multiple effects within a single monad. This modularity-via-integration, however, has consequences for formal verification: because its effects are all encapsulated within the single monad, they are not distinguished syntactically within the type system of a specification language itself. Wadler [109] "married" effect types to monads, and previous work by the authors [108] seems to be the first marriage of effect types to layered monads. This latter marriage seems to be important for exploiting monadic semantics in formal methods: layered monads provide a modular semantics of effects including by-construction properties and effect types allow the expression of these properties in a formal proof system like Coq (e.g., Figure 3.14).

As a concept for formal (i.e., machine-checked) verification, monads are less common, although not unheard of [22, 72, 97, 94] and the use of both effect types and layered monads distinguishes the current work from these. Furthermore, ReWire's monad language

includes the reactive resumption monad transformer, which does not appear to have been formalized previously.

3.3 Background: ReWire's Programming Model

The purpose of this section is twofold: (1) to make this article as self-contained as possible by providing sufficient background on ReWire and (2) to motivate RWC's type system and operational semantics. Throughout this section, we explicitly link this background material to subsequent sections on RWC. ReWire is a subset of Haskell and uses ideas from monadic semantics as an organizing principle of the language. It is, therefore, assumed of necessity that the reader is, at least, somewhat familiar with functional programming and monads.

ReWire is a subset of the Haskell functional programming language [78]—i.e., ReWire programs are Haskell programs, but not necessarily *vice versa*. All ReWire programs can be compiled to synthesizable VHDL with the ReWire compiler. The principal difference between Haskell and ReWire is that recursion in ReWire is restricted to tail recursion so that every ReWire program requires only a finite, bounded memory footprint. Unbounded recursion requires a stack or heap for compilation and such unbounded structures are anathema to hardware's fixed storage.

ReWire has type constructors for devices where a *device* represents a clocked computation that, for each clock cycle, takes an input of type `i`, produces an output of type `o`, and may possess internal storage of type `s` (see Fig. 3.1). The type of `d` as shown would be `d :: ReT i o (StT s Id) ()`, where `ReT` and `StT` are the reactive resumption and state monad transformers and `Id` is the identity monad (about all of which we say more below in the next sections). Device `d` is clocked, as illustrated in the inset figure, although the clock is represented by the underlying structure of reactive resumptions rather than as

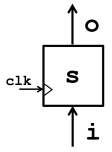


Figure 3.1: Device d.

an explicit parameter. A device is created in ReWire by either iterating a function or through composition of existing devices. Previous work [47] introduced operators for constructing devices and composing them into larger, interconnected devices; Section 3.3.3 presents a simple device specification template in ReWire.

3.3.1 Background: Monads

A monad is a triple $\langle \mathtt{M}, \mathsf{return}, \mathsf{>>=} \rangle$ consisting of a type constructor $\mathtt{M}$ and two operations:

$$\begin{aligned} &\mathsf{return} \;:\; \mathtt{a} \to \mathtt{M}\,\mathtt{a} && \text{— “unit”} \\ &(\mathsf{>>=}) \;:\; \mathtt{M}\,\mathtt{a} \to (\mathtt{a} \to \mathtt{M}\,\mathtt{b}) \to \mathtt{M}\,\mathtt{b} && \text{— “bind”} \end{aligned}$$

These operations must obey the well-known *monad laws* [69, 60] (these are (Left-Unit), (Right-Unit), and (associativity) in Figure 3.14). The return operator is the monadic analogue of the identity function, injecting a value into the monad. The >>= operator is a form of sequential application. The “null bind” operator, $\mathtt{>>} \;:\; \mathtt{M}\,\mathtt{a} \to \mathtt{M}\,\mathtt{b} \to \mathtt{M}\,\mathtt{b}$, is defined as: $\mathtt{x} \mathtt{>>} \mathtt{k} \;=\; \mathtt{x} \mathtt{>>=} \lambda\_.\mathtt{k}$. The binding (i.e., “$\lambda\_$”) acts as a dummy variable, ignoring the value produced by x.

3.3.2 Background: Monad Transformers

The organizing principle underlying ReWire are *reactive resumption monads with state* [44] (RRMS), which encapsulate a notion of computation appropriate to hardware—namely, synchronous parallelism. RRMS support the expression of structural hardware designs in a functional style [47]. RWC is a computational λ-calculus whose syntax and semantics formalizes RRMS in Coq. In particular, RWC's type system includes constructors that correspond to the state and reactive resumption monad transformers. For the sake of being self-contained, we provide the reader with Haskell definitions of the StT and ReT monad transformers. This code is meant only to aid the reader in comprehending the

intended semantics of RWC. If more background is required on RRMS, please consult the references [44, 83].

State Monad Transformer

The state monad transformer is a well-documented structure in functional programming and semantics [60]. The Haskell code for the state monad transformer, `StT`, along with its `lifting` functions is below:

```
data StT s m a = StT (s -> m (a,s))
lift_StT :: m a -> StT s m a
lift_StT m = StT (\ s -> m >>=_m \ v -> return_m (v,s))
get :: StT s m s
get            = StT (\ s -> return_m (s,s))
put :: s -> StT s m ()
put s          = StT (\ _ -> return_m ((),s))
```

The `lift` converts an `m a` computation into an `StT s m a` computation. The `get` operation returns the current value of the `s`-store while the `put s` operation replaces the current store with store `s`. In the definitions above, the binds and returns for the `m` monad are affixed with a subscript to disambiguate them from the operations being defined.

Reactive Resumption Monad Transformer

Computations in `ReT i o m a` may be viewed intuitively as (potentially infinite) sequences of `m` computations. If that sequence terminates, it produces an `a`-value, otherwise it produces an `o`-output value and a continuation. Both `lift` operations convert an `m` computation into respective enriched computations. Computations in `ReT` over layered state monads correspond closely to synchronous hardware as discussed in previous work [83]. The Haskell code for the reactive resumption monad transformer, `ReT`, along with its associated functions is below:

```
data ReT i o m a = Pause (m (Either a (o,i -> ReT i o m a)))
```

```
liftReT :: m a -> ReT i o m a
liftReT m = Pause (m >>=m returnm . Left)
signal :: o -> ReT i o m i
signal o = Pause (returnm (Right (o,returnm . returnReT)))
data Either a b = Left a | Right b
```

Recall that function composition (i.e., "`.`") and sum types (i.e., `Either`) are built-in to Haskell. In terms of the device `d` example above, the operation `signal o` represents the end of a clock cycle and sets the output signal of `d` to `o`. RWC includes a `pause` primitive in the term syntax (Fig. 3.3) as a means of representing `signal`.

By-construction Properties of Layered Monads

Layered state monads have multiple `StT` applications—e.g., $\texttt{M} = \texttt{StT}\ \texttt{s}_1\ (\texttt{StT}\ \texttt{s}_2\ \texttt{Id})$ is a two-layer state monad. They have a number of useful properties by construction [45], including:

$$\texttt{put}\ \texttt{s}' \gg \texttt{put}\ \texttt{s} \quad = \texttt{put}\ \texttt{s}$$

$$\texttt{put}\ \texttt{s} \gg \texttt{lift}_{\texttt{StT}}\ \varphi = \texttt{lift}_{\texttt{StT}}\ \varphi \gg \texttt{put}\ \texttt{s}$$

The first rule is an intra-layer property (a.k.a., "clobber") while the second is an inter-layer property (a.k.a., "atomic non-interference"). Clobber states that the `put s` cancels earlier effects on the same layer. By convention for a fixed state $\texttt{s}_0$, we define $\texttt{mask} = \texttt{put}\ \texttt{s}_0$; the `mask` included in the term syntax of RWC generalizes this idea. Atomic non-interference states that effects from different state layers commute. The equational logic derived in Coq for RWC presented in Section 3.6 gives generalizations of both properties.

3.3.3 Defining Devices in ReWire

Simple ReWire devices are generally defined as tail recursive functions whose codomain is written in terms of the `ReT` layer. Assume functions, $\texttt{internal} :: \texttt{i} \to \texttt{StT}\ \texttt{s}\ \texttt{Id}\ \texttt{v}$ and $\texttt{external} :: \texttt{i} \to \texttt{v} \to \texttt{o}$, are defined which specify the internal and external behaviors

of device d. Function internal takes the input i, performs some computation with the current internal storage s, and produces an intermediate result v. Function external takes the input i and the result v and produces the next output signal for d.

Given an initial input i_0, $d = \texttt{dev}\ i_0$ where corecursive function dev is defined as:

```
dev :: i -> ReT i o (StT s Id) ()
dev i = lift_ReT (internal i)  >>= \ v ->
signal (external i v) >>= \ i' ->
dev i'
```

At the beginning of a clock cycle, dev first consumes input, i, performs internal i computation on the internal storage s, and then outputs the external i v signal at the end of the cycle.

Device definitions are expressed with an explicit corecursion operator, unfold; for example, the device dev above would be written:

```
unfold i_0 (\ i -> internal i >>= \ v -> return (Right (
    external i v, id)))
```

For this reason, Figure 3.3 includes syntax for an unfold primitive and its semantics are defined in subsequent sections.

3.3.4 Background: Goguen-Meseguer Non-interference

The essence of the Goguen-Meseguer noninterference information flow model [39] and its many descendants is that systems, broadly construed, are state machines whose inputs and outputs are partitioned by security level. The definition of information flow is formulated in terms of sequences of stateful operations of mixed security levels and stipulates that high-level operations must not affect low-level outputs. More concretely, for any mixed-level sequence, $s = (l_1 ; h_1 ; \ldots ; l_n ; h_n)$, the low-level outputs of s must be identical to those produced by $(l_1 ; \ldots ; l_n)$, which is the result of filtering out from s all high-level operations.

3.3.5 Marrying Effects & Layered State Monads

"By construction" properties of layered state monads [45] tell us that high- and low-security operations commute (a.k.a., atomic non-interference) and that mask_H cancels high-level operations (i.e., $\varphi_H \gg \mathsf{mask}_H = \mathsf{mask}_H$). This cancelling property is known as the "clobber rule" [45]. The atomic non-interference and clobber rules are helpful in demonstrating that monadic noninterference equations (like that of the previous section) hold for particular software and hardware applications [45, 83].

The Goguen-Meseguer model was recast in monadic terms previously [45], so that high-level effects must be cancellable without affecting the low-level effects. Here, the utility of the RWC effect type system becomes evident, because it can statically distinguish computations occurring on distinct layers. For the sake of concreteness, consider the case of a monad, M, with a high- and low-security stores types, H and L. High and low operations may be distinguished by the RWC effect type system by annotating the layers with effect labels:

$$\varphi_H : \mathsf{StT}\ \mathsf{RW}\ H\ (\mathsf{StT}\ \langle\rangle\ L\ \mathsf{Id})() \qquad \varphi_L : \mathsf{StT}\ \langle\rangle\ H\ (\mathsf{StT}\ \mathsf{RW}\ L\ \mathsf{Id})()$$

Note that φ_H (resp., φ_L) only has read-write effects (RW) on the outer (resp., inner) state layer of M. Furthermore, we assume the existence of an operation, mask_H which initializes the H state layer. The mask_H operation can be assumed to be $\mathsf{put}\ \mathsf{s}_0$ on the H-layer, where s_0 is an arbitrary, fixed value in H. Then, the monadic formulation of non-interference boils down to demonstrating that equations like the following hold: $\varphi_H \gg \varphi_L \gg \mathsf{mask}_H = \varphi_L \gg \mathsf{mask}_H$. This means that reinitializing the H layer cancels the effects of high-security operations like φ_H. This is the monadic analogy of Goguen and Meseguer's filtering out of high-security operations.

$$
\begin{aligned}
\ell \in \mathit{EffectLabel} &::= \langle\rangle \mid \mathsf{R} \mid \mathsf{W} \mid \mathsf{RW} \\
\mathsf{S} \in \mathit{StateMonad} &::= \mathsf{Id} \mid \mathsf{StT}\ \ell\ \tau\ \mathsf{S} \\
\mathsf{M} \in \mathit{Monad} &::= \mathsf{S} \mid \mathsf{ReT}\ \tau\ \tau'\ \mathsf{S} \\
\tau, \tau' \in \mathit{Type} &::= \tau \to \tau' \mid \tau \times \tau' \mid \tau + \tau' \mid () \mid \mathsf{M}\ \tau
\end{aligned}
$$

Figure 3.2: Syntax of RWC types

3.4 RWC: The ReWire Core Calculus

This section introduces the syntax (Section 3.4.1), type system (Section 3.4.2) and operational semantics (Section 3.4.3) of the ReWire Calculus (RWC). RWC is a computational λ-calculus that extends the functional features of a typed lambda calculus with support for stateful effects and reactive parallelism. These effects are encapsulated through the use of *monads* [69], enabling us to provide a useful equational theory in the presence of effects. The addition of effects to a computational λ-calculus was examined in [109].

3.4.1 Syntax

This section introduces the syntax of RWC, which is a variety of computational λ-calculus extended with operations for synchronous, stateful parallelism. Here, the stateful component is organized as *layered* state monads—i.e., monads created by multiple applications of the state monad transformer. Layered state monads have by-construction properties that support information flow security verification [45, 83]; we defer presenting the general formalization of these by-construction properties until Section 3.6. Section 3.3 provides the reader with some background on monad transformers, although readers requiring more should consult the references.

Types

Figure 3.2 shows the syntax of types. As a computational λ-calculus, RWC extends the

simply-typed λ-calculus with unit, sum, and product types along with a notion of *computational* types: if M is a monad and τ is a type, then $M\ \tau$ is the type of computations in the monad M with a result value of type τ. Exactly which monad stands in for M will determine what sort of computational effects are possible. RWC permits the use of monads built in terms of the Id (identity) monad and the ReT (reactive resumption), and StT (state) monad transformers, where ReT must be the outermost monad transformer application (if it is present). RWC's monads encompass the combination of resumption and layered state monads found in [44] with the addition of *effect labels* attached to each StT. The presence of an effect label ℓ at a given layer certifies that the computation has at *most* the effects ℓ at that layer. For example, the effect label W reflects the *possibility* that a computation will write, not the necessity, and certifies that the computation will *not* read.

We note in passing that the *denotational* semantics of these monads corresponds exactly to the semantics of their Haskell equivalents, up to the erasure of the effect labels and with the considerable simplification that lifted domains are not necessary due to the absence of general recursion; see [82] for further details.

Terms

Figure 3.3 shows the syntax of terms. Note the widespread use of type and monad subscripts. These are necessary to ensure that every term has a unique type, and to handle overloading of monadic operations. We will sometimes omit these subscripts, as long as doing so does not introduce ambiguity.

We will not remark on the standard λ-calculus machinery, other than to note that the constructs used for destructing pairs and elements of sum type are slightly nonstandard. The term constructor proj, used for destructing pairs, takes two subterms: the first corresponding to the pair being deconstructed—suppose it has type $\tau \times \tau'$—and the second corresponding to a *function* of type $\tau \to \tau' \to \tau''$ that produces a value from the pair's

$$
\begin{aligned}
\mathit{Identifier} &::= x \mid y \mid z \mid w \mid \textit{etc.} \\
t \in \mathit{Term} &::= x \mid t\ t' \mid \lambda x : \tau.t \mid () \mid \langle t, t' \rangle \mid \mathsf{proj}\ t\ t' \mid \\
&\quad \mid \mathsf{inl}_{\tau}\ t \mid \mathsf{inr}_{\tau}\ t \mid \mathsf{case}\ t\ t'\ t'' \mid \mathsf{return}_{\mathsf{M}}\ t \mid t \mathbin{\texttt{>>=}} t' \\
&\quad \mid \mathsf{lift}_{\mathsf{M}}\ t \mid \mathsf{elevate}_{\mathsf{S}}\ t \mid \mathsf{get}_{\mathsf{S}} \mid \mathsf{put}\ t \mid \mathsf{pause}_{\mathsf{M},\tau}\ t \\
&\quad \mid \mathsf{runSt}\ t\ t' \mid \mathsf{runId}\ t \mid \mathsf{unfold}_{\mathsf{M},\tau,\tau'}\ t\ t' \mid \mathsf{runRe}_{\tau}\ t \\
v, v' \in \mathit{Value} &::= \lambda x : \tau.t \mid () \mid \langle v, v' \rangle \mid \mathsf{inl}_{\tau}\ v \mid \mathsf{inr}_{\tau}\ v \mid \mathsf{return}_{\mathsf{M}}\ v \\
&\quad \mid v \mathbin{\texttt{>>=}} v' \mid \mathsf{lift}_{\mathsf{M}}\ v \mid \mathsf{elevate}_{\mathsf{M}}\ v \mid \mathsf{get}_{\mathsf{S}} \mid \mathsf{put}\ v \\
&\quad \mid \mathsf{pause}_{\mathsf{M},\tau}\ v \mid \mathsf{runSt}\ v\ v' \mid \mathsf{runRe}_{\tau}\ v \\
&\quad \mid \mathsf{unfold}_{\mathsf{M},\tau,\tau'}\ v\ v' \\
\Sigma \in \mathit{Store} &::= \mathsf{nil} \mid s :: \Sigma \\
c \in \mathit{Config} &::= \langle t, \Sigma \rangle \\
D \in \mathit{DoneConfig} &::= \langle\, \mathsf{return}_{\mathsf{M}}\ v, \Sigma \rangle \mid \langle\, \mathsf{pause}_{\mathsf{M},\tau}\ v, \Sigma \rangle
\end{aligned}
$$

Figure 3.3: Syntax of terms, stores, and configurations

elements. (Note that the conventional left- and right-projection operators can be constructed in terms of the proj operator.) The term constructor case, used for destructing elements of sum type, takes three subterms: the first is the scrutinee of type $\tau + \tau'$, the second to a function f_1 of type $\tau \to \tau''$, and the third to a function f_2 of type $\tau' \to \tau''$. If the scrutinee evaluates to inl v (resp., inr v), then v will be passed to f_1 (resp., f_2).

Computations are defined in terms of certain primitives. The (overloaded) term constructors return and >>= correspond respectively to the unit and bind operations of the monads, and lift to the lift operation of each monad transformer. Terms typed in a state monad may read and write to the store using the get and put operations. The term constructor elevate adds effect labels—e.g., W or R— to the effect labels, if any, on a state monad computation; thereby, converting state monad computations with a less permissive types to a more permissive type (where "permissiveness" is understood as in Figure 3.5). For example, a term t of type StT R τ Id τ' can be typecast into the more permissive type StT RW τ Id τ' via elevate, essentially de-certifying that t does not write. (A cast in the "other direction", to StT $\langle\rangle$ τ Id τ', is not permitted by the type system.)

Reactive computations are defined in terms of the primitives pause and unfold. The term pause t is essentially a suspended computation that is waiting for an input value, and unfold can be used to produce "looping" computations; we postpone a discussion of their exact semantics until we have discussed the type system in greater detail. Finally, the term constructors runRe, runSt, and runId allow the effects of a given monad transformer to be reflected into the base monad. It may be helpful to view runRe as executing a single step of a resumption-monadic computation, runSt as supplying the initial state for the uppermost state layer, and runId as moving from the effect-free Id monad into the universe of non-monadic terms.

Stores and Configurations.

Figure 3.3 (bottom) shows the syntax of *stores* and *configurations*, which will be used to specify the semantics of computations. A store is a list of terms, each of which corresponds semantically to a state monad transformer, and a configuration $\langle t, \Sigma \rangle$ pairs a term t with a state Σ. Generally, we use the metavariables s, s', s'' to refer store values.

3.4.2 Type System

Typing rules for terms are given in Figure 3.4. Typing judgments take the form $\Gamma \vdash t : \tau$, where Γ is a set of assumptions (i.e., a mapping of variables to types). For the empty context, we write $\{\}$. Many of the rules are standard, reflecting the rules of computational λ-calculus. The rules for get, put, and elevate require special attention, as they directly involve effect labels. Rule T-Get restricts the effect label on the top monad transformer to include a read label, and T-Put restricts it to include a write label. These restrictions are expressed in terms of an ordering on effect labels (which is really nothing more than the subset relation) given in Figure 3.5 at left. For T-Elevate, we require that the target monad S' has (non-strictly) more effect labels than the source monad S; its precise meaning is expressed in Figure 3.5 at right. The elevate operation permits us to *decertify* that a

$$\frac{}{\Gamma, x:\tau \vdash x:\tau}\ (\textsc{Var}) \qquad \frac{\Gamma, x:\tau \vdash t:\tau'}{\Gamma \vdash \lambda x:\tau.t:\tau \to \tau'}\ (\textsc{Abs}) \qquad \frac{\Gamma \vdash t':\tau \to \tau' \quad \Gamma \vdash t:\tau}{\Gamma \vdash t'\ t:\tau'}\ (\textsc{App})$$

$$\frac{\Gamma \vdash t:\tau}{\Gamma \vdash \mathsf{inl}_{\tau'}\ t:\tau+\tau'}\ (\textsc{Inl}) \qquad \frac{\Gamma \vdash t:\tau}{\Gamma \vdash \mathsf{inr}_{\tau'}\ t:\tau'+\tau}\ (\textsc{Inr}) \qquad \frac{\Gamma \vdash t:\tau \quad \Gamma \vdash t':\tau'}{\Gamma \vdash \langle t, t' \rangle : \tau \times \tau'}\ (\textsc{Pair})$$

$$\frac{}{\Gamma \vdash () : ()}\ (\textsc{Unit}) \qquad \frac{\Gamma \vdash t:\tau \times \tau' \quad \Gamma \vdash t':\tau \to \tau' \to \tau''}{\Gamma \vdash \mathsf{proj}\ t\ t' : \tau''}\ (\textsc{Proj})$$

$$\frac{\Gamma \vdash t:\tau' + \tau'' \quad \Gamma \vdash t':\tau' \to \tau \quad \Gamma \vdash t'':\tau'' \to \tau}{\Gamma \vdash \mathsf{case}\ t\ t'\ t'' : \tau}\ (\textsc{Case})$$

$$\frac{\Gamma \vdash t:\tau}{\Gamma \vdash \mathsf{return}_{\mathsf{M}}\ t:\ \mathsf{M}\ \tau}\ (\textsc{Return}) \qquad \frac{\Gamma \vdash t:\ \mathsf{M}\ \tau \quad \Gamma \vdash t':\tau \to \mathsf{M}\ \tau'}{\Gamma \vdash t\ \texttt{>>=}\ t':\ \mathsf{M}\ \tau'}\ (\textsc{Bind})$$

$$\frac{\Gamma \vdash t : \mathsf{S}\ \tau}{\Gamma \vdash \mathsf{lift}_{(\mathsf{StT}\ell\tau'\mathsf{S})}\ t : \mathsf{StT}\ \ell\ \tau'\ \mathsf{S}\ \tau}\ (\textsc{LiftSt})$$

$$\frac{\Gamma \vdash t:\ \mathsf{S}\ \tau}{\Gamma \vdash \mathsf{lift}_{(\mathsf{ReT}\tau'\tau''\mathsf{S})}\ t:\ \mathsf{ReT}\ \tau'\ \tau''\ \mathsf{S}\ \tau}\ (\textsc{LiftRe})$$

$$\frac{\mathsf{R} \leq \ell}{\Gamma \vdash \mathsf{get}_{(\mathsf{StT}\ell\tau\mathsf{S})} : \mathsf{StT}\ \ell\ \tau\ \mathsf{S}\ \tau}\ (\textsc{get}) \qquad \frac{\Gamma \vdash t : \tau \quad \mathsf{W} \leq \ell}{\Gamma \vdash \mathsf{put}\ \ t : \mathsf{StT}\ \ell\ \tau\ \mathsf{S}\ ()}\ (\textsc{put})$$

$$\frac{\Gamma \vdash t:\ \mathsf{StT}\ \ell\ \tau'\ \mathsf{S}\ \tau \quad \Gamma \vdash t':\ \tau'}{\Gamma \vdash \mathsf{runSt}\ \ t\ t':\ \mathsf{S}\ (\tau \times \tau')}\ (\textsc{RunSt}) \qquad \frac{\Gamma \vdash t:\ \mathsf{Id}\ \tau}{\Gamma \vdash \mathsf{runId}\ t:\ \tau}\ (\textsc{RunId})$$

$$\frac{\Gamma \vdash t : \mathsf{S}\ (\tau' \times (\tau \to \mathsf{ReT}\ \tau\ \tau'\ \mathsf{S}\ \tau''))}{\Gamma \vdash \mathsf{pause}_{(\mathsf{ReT}\tau\tau'\mathsf{S},\tau'')}\ \ t : \mathsf{ReT}\ \tau\ \tau'\ \mathsf{S}\ \tau''}\ (\textsc{pause})$$

$$\frac{\Gamma \vdash t : \tau''' \quad \Gamma \vdash t' : \tau''' \to \mathsf{S}\ (\tau''+(\tau' \times (\tau \to \tau''')))}{\Gamma \vdash \mathsf{unfold}_{(\mathsf{ReT}\ \tau\ \tau'\ \mathsf{S},\tau'',\tau''')}\ \ t\ t' : \mathsf{ReT}\ \tau\ \tau'\ \mathsf{S}\ \tau''}\ (\textsc{unfold})$$

$$\frac{\Gamma \vdash t : \mathsf{ReT}\ \tau\ \tau'\ \mathsf{S}\ \tau''}{\Gamma \vdash \mathsf{runRe}_{\tau}\ t : \mathsf{S}\ (\tau''+(\tau' \times (\tau \to \mathsf{ReT}\ \tau\ \tau'\ \mathsf{S}\ \tau')))}\ (\textsc{RunRe})$$

$$\frac{\Gamma \vdash t : \mathsf{S}\ \tau \quad \mathsf{S} \leq \mathsf{S}'}{\Gamma \vdash \mathsf{elevate}_{\mathsf{S}'}\ \ t : \mathsf{S}'\ \tau}\ (\textsc{Elevate})$$

Figure 3.4: Typing judgments for terms.

computation does *not* read or write at any given state layers, but not to remove existing effect labels.

Stores and configurations also have a notion of type, defined by the rules of Figure 3.6. A store Σ is said to *match* a monad M if the types of its elements correspond, in order, to the state types of the state monad transformers in M. For this, we simply write that Σ matches M. A configuration $\langle t, \Sigma \rangle$, then, has type $M\ \tau$ if and only if Σ matches M and $\{\} \vdash t : M\ \tau$. We write this $\langle t, \Sigma \rangle \triangleright M\ \tau$. A simple, yet useful property of our type system is that every term (resp. configuration) has a unique type, as stated in Theorem 3.1.

Theorem 3.1 (Uniqueness of Types)**.** *If* $\Gamma \vdash t : \tau$ *and* $\Gamma \vdash t : \tau'$*, then* $\tau = \tau'$*. Also, if* $\langle t, \Sigma \rangle \triangleright \tau$ *and* $\langle t, \Sigma \rangle \triangleright \tau'$*, then* $\tau = \tau'$*.*

Furthermore, substitutions preserve typing judgments. To see this, we need to define substitution and collect some facts about free variables, substitutions and types. For any term t, the set of free variables in t, $FV(t)$, is defined as follows:

$$
\begin{array}{rcll}
FV(x) & = & \{x\} & \\
FV(t\,u) & = & FV(t) \bigcup FV(u) & \\
FV(\lambda x : \tau . t) & = & FV(t) \setminus \{x\} & \\
FV(c_0) & = & \{\}, & \text{for any nullary term } c_0 \\
FV(c_n\,t_1, \ldots, t_n) & = & \bigcup_{i=1}^{n} FV(t_i), & \text{for an } n\text{-ary term } c_n
\end{array}
$$

In the last clause above, the c in '$c\,t_1, \ldots, t_n$' stands for term constructors such as case, return_M, etc. If $FV(t) = \emptyset$, then t is said to be *closed*. We now define the substitution

of v for free occurrences of x in t, written '$t[x := v]$', thusly:

$$
\begin{array}{rcll}
x[x := v] & = & v & \\
y[x := v] & = & y & \text{if } y \neq x \\
(t\,u)[x := v] & = & (t[x := v])(u[x := v]) & \\
(\lambda x : \tau.t)[x := v] & = & (\lambda x : \tau.t) & \\
(\lambda y : \tau.t)[x := v] & = & \lambda y : \tau.(t[x := v]) & \text{if } y \neq x \text{ and } y \notin FV(v) \\
(c\,t_1, \ldots, t_n)[x := v] & = & (c\,(t_1[x := v]), \ldots, (t_n[x := v])) &
\end{array}
$$

This definition of substitution preserves typing judgments. This requires that if free variables occur in well-typed terms, then there must be a typing assignment for those variables relative to the context. As stated in Lemma 3.2.

Lemma 3.2. *If $x \in FV(t)$ and $\Gamma \vdash t : \tau$, then there exists τ' such that $x : \tau' \in \Gamma$.*

From this Corollary 3.3 follows—namely, that a term is closed if it it is well-typed in the empty context.

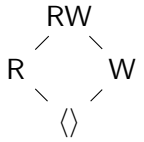
Corollary 3.3. *If $\{\} \vdash t : \tau$, then t is closed.*

Moreover, we have Lemma 3.4 as a consequence—that the context of a typing judgment does not alter typing judgments, so long as all each context maintains assignments of types to any free variables.

Lemma 3.4. *If $\Gamma \vdash t : \tau$ and, if, for all x such that $x \in \mathbf{FV}(t), \Gamma, x = \Gamma', x$, then $\Gamma' \vdash t : \tau$.*

Finally, we have Theorem 3.5—that is, it follows that substitution preserves typing judgments.

Theorem 3.5. *If $x : \tau', \Gamma \vdash t : \tau$ and $\{\} \vdash v : \tau'$, then $\Gamma \vdash (t[x := v]) : \tau$.*



$$\frac{}{\mathsf{Id} \leq \mathsf{Id}}\ \text{(L-Id)} \qquad \frac{\ell \leq \ell' \quad S \leq S'}{\mathsf{StT}\ \ell\ \tau\ S \leq \mathsf{StT}\ \ell'\ \tau\ S'}\ \text{(L-StT)}$$

Figure 3.5: Ordering on effect labels (given by the diagram) and on state monads.

$$\frac{\Sigma \text{ matches } \mathsf{S}}{\Sigma \text{ matches } \mathsf{ReT}\ \tau\ \tau'\ \mathsf{S}}\ \text{(M-ReT)} \qquad \frac{\{\} \vdash s : \tau \quad \Sigma \text{ matches } \mathsf{S}}{(s :: \Sigma) \text{ matches } \mathsf{StT}\ \ell\ \tau\ \mathsf{S}}\ \text{(M-StT)}$$

$$\frac{\{\} \vdash t : \mathsf{M}\ \tau \quad \Sigma \text{ matches } \mathsf{M}}{\langle t, \Sigma \rangle \triangleright \mathsf{M}\ \tau}\ \text{(T-Config)}$$

Figure 3.6: Typing judgments for stores (top) and configurations (bottom).

3.4.3 Small-Step Operational Semantics

In this section we describe the semantics of RWC in a small-step operational style. As a computational λ-calculus, RWC contains both functional features (functional abstraction and application) as well as effectful ones (mutable state and reactive parallelism). The operational semantics is structured around this dichotomy, with two interdefined notions of reduction: *pure* and *effectful* reduction. Pure reduction reflects the notion of effect-free evaluation. A pure reduction judgment takes the form $t \rightsquigarrow t'$; note that this makes no mention of any store. Effectful reduction provides semantics to computational terms which may have effects. Thus an effectful reduction judgment takes the form $\langle t, \Sigma \rangle \rightsquigarrow \langle t', \Sigma' \rangle$.

The rules for pure and effectful reduction are given in Figures 3.7 and 3.8, respectively. We adopt a call-by-value evaluation strategy, as this is (we feel) simpler to work with metatheoretically than call-by-name or -need. This may seem strange in light of ReWire's antecedents in Haskell (which is a non-strict language), but since ReWire is a strongly normalizing subset of Haskell, it does not matter whether we choose an eager or lazy evaluation strategy from a "backwards compatibility" point of view: since there is no "bottom" value, strictness is not a concern.As stated in Theorem 3.6, the reduction relation defined by the rules for pure and effectful reduction is deterministic.

Theorem 3.6. *If* $t \rightsquigarrow t'$ *and* $t \rightsquigarrow t''$*, then* $t' = t''$*. Also, if* $\langle t, \Sigma \rangle \rightsquigarrow \langle t', \Sigma' \rangle$ *and* $\langle t, \Sigma \rangle \rightsquigarrow \langle t'', \Sigma'' \rangle$*, then* $\langle t', \Sigma' \rangle = \langle t'', \Sigma'' \rangle$.

A few of the rules require close inspection. To begin with, we note that pure and effectful reduction are interdefined. Rule STM-ST of Figure 3.8 allows pure reduction to be "lifted" into the universe of effectful reduction: if the term component t of a configuration $\langle t, \Sigma \rangle$ still has not been evaluated to a value, we will continue to evaluate it without changing the store. Dually, if less obviously, the rule ST-RunIdMo in Figure 3.7 allows monadic evaluation in the identity monad (and *only* in the identity monad) to be reified in a pure setting. If we wish to run a computation in a more complex monad, we may use runRe and runSt to "peel off" one monad transformer at a time, until we reach the Id monad at the core. In the runSt case, we must supply an initial value for the corresponding state layer, producing a computation in the base monad which will return the post-value for that layer. The runRe operator will produce a computation in the base monad that either returns a final result value, or an output value paired with a continuation waiting on more input.

Note also the interaction between the rule STM-LiftSt, STM-Get, and STM-Put. The get and put operations always operate on the 'head' (leftmost) item in the store. Applying $\mathsf{lift}_{\mathsf{StT}}$ to these operations allows us to access items deeper in the store, by executing the underlying computation against the "tail" of the store and leaving the "head" item unchanged.

The rule STM-Unfold may be justified directly by the Haskell definition of unfold. Rule STM-Pause is more subtle. The basic idea, however, is that if a pause arises to the left of a >>=, we should "absorb" what comes to the right of the >>= into the pause's continuation, guaranteeing that we make progress towards a "done" configuration.

$$\frac{}{(\lambda x : \tau.t)v \rightsquigarrow t[x := v]}\ \text{(ST-AppAbs)} \qquad \frac{t \rightsquigarrow t''}{t\ t' \rightsquigarrow t''\ t'}\ \text{(ST-App1)} \qquad \frac{t \rightsquigarrow t'}{v\ t \rightsquigarrow v\ t'}\ \text{(ST-App2)} \qquad \frac{t \rightsquigarrow t''}{\langle t, t' \rangle \rightsquigarrow \langle t'', t' \rangle}\ \text{(ST-Pair1)}$$

$$\frac{t \rightsquigarrow t'}{\langle v, t \rangle \rightsquigarrow \langle v, t' \rangle}\ \text{(ST-Pair2)} \qquad \frac{t \rightsquigarrow t'}{\mathsf{proj}\ t\ t'' \rightsquigarrow \mathsf{proj}\ t'\ t''}\ \text{(ST-Proj1)} \qquad \frac{t \rightsquigarrow t'}{\mathsf{proj}\ v\ t \rightsquigarrow \mathsf{proj}\ v\ t'}\ \text{(ST-Proj2)}$$

$$\frac{}{\mathsf{proj}\ \langle v, v' \rangle\ v'' \rightsquigarrow (v''\ v)\ v'}\ \text{(ST-Proj)} \qquad \frac{t \rightsquigarrow t'}{\mathsf{inl}_T\ t \rightsquigarrow \mathsf{inl}_T\ t'}\ \text{(ST-Inl)} \qquad \frac{t \rightsquigarrow t'}{\mathsf{inr}_T\ t \rightsquigarrow \mathsf{inr}_T\ t'}\ \text{(ST-Inr)}$$

$$\frac{t \rightsquigarrow t'''}{\mathsf{case}\ t\ t'\ t'' \rightsquigarrow \mathsf{case}\ t'''\ t'\ t''}\ \text{(ST-Case1)} \qquad \frac{t \rightsquigarrow t''}{\mathsf{case}\ v\ t\ t' \rightsquigarrow \mathsf{case}\ v\ t''\ t'}\ \text{(ST-Case2)} \qquad \frac{t \rightsquigarrow t'}{\mathsf{case}\ v\ v'\ t \rightsquigarrow \mathsf{case}\ v\ v'\ t'}\ \text{(ST-Case3)}$$

$$\frac{}{\mathsf{case}\ (\mathsf{inl}_T\ v)\ v'\ v'' \rightsquigarrow v'\ v}\ \text{(ST-CaseL)} \qquad \frac{}{\mathsf{case}\ (\mathsf{inr}_T\ v)\ v'\ v'' \rightsquigarrow v''\ v}\ \text{(ST-CaseR)} \qquad \frac{\langle v, \mathsf{nil} \rangle \rightsquigarrow \langle t, \mathsf{nil} \rangle}{\mathsf{runId}\ v \rightsquigarrow \mathsf{runId}\ t}\ \text{(ST-RunIdMo)}$$

Figure 3.7: Reduction Rules for Lambda Calculus Reduction. These rules specify a call-by-value evaluation strategy on RWC.

$$\frac{t \rightsquigarrow t'}{\langle t, \Sigma \rangle \rightsquigarrow \langle t', \Sigma \rangle}\ \text{(STM-ST)} \qquad \frac{\langle v, \Sigma \rangle \rightsquigarrow \langle t, \Sigma' \rangle}{\langle v\ \texttt{>>=}\ v', \Sigma \rangle \rightsquigarrow \langle t\ \texttt{>>=}\ v', \Sigma' \rangle}\ \text{(STM-Bind)} \qquad \frac{}{\langle\ \mathsf{return_M}\ v\ \texttt{>>=}\ v', \Sigma \rangle \rightsquigarrow \langle v'\ v, \Sigma \rangle}\ \text{(STM-BindRet)}$$

$$\frac{\langle v, \Sigma \rangle \rightsquigarrow \langle t, \Sigma' \rangle}{\langle \mathsf{lift}_{(\mathsf{StT}\ \ell\ \tau\ \mathsf{S})}\ v, s :: \Sigma \rangle \rightsquigarrow \langle \mathsf{lift}_{(\mathsf{StT}\ \ell\ \tau\ \mathsf{S})}\ t, s :: \Sigma' \rangle}\ \text{(STM-LiftSt)} \qquad \frac{\langle v, \Sigma \rangle \rightsquigarrow \langle t, \Sigma' \rangle}{\langle \mathsf{lift}_{(\mathsf{ReT}\ \tau\ \tau'\ \mathsf{S})}\ v, \Sigma \rangle \rightsquigarrow \langle \mathsf{lift}_{(\mathsf{ReT}\ \tau\ \tau'\ \mathsf{S})}\ t, \Sigma' \rangle}\ \text{(STM-LiftRe)}$$

$$\frac{}{\langle \mathsf{lift_M}\ (\mathsf{return}_{\mathsf{M}'}\ v), \Sigma \rangle \rightsquigarrow \langle \mathsf{return_M}\ v, \Sigma \rangle}\ \text{(STM-LiftRet)}$$

$$\frac{}{\langle \mathsf{get_S}, s :: \Sigma \rangle \rightsquigarrow \langle \mathsf{return_S}\ s, s :: \Sigma \rangle}\ \text{(STM-Get)} \qquad \frac{}{\langle \mathsf{put}\ v, s :: \Sigma \rangle \rightsquigarrow \langle \mathsf{return_S}\ (), v :: \Sigma \rangle}\ \text{(STM-Put)}$$

$$\frac{\langle t, \Sigma \rangle \rightsquigarrow \langle t', \Sigma' \rangle}{\langle \mathsf{elevate_S}\ t, \Sigma \rangle \rightsquigarrow \langle \mathsf{elevate_S}\ t', \Sigma' \rangle}\ \text{(STM-Elevate)} \qquad \frac{}{\langle \mathsf{elevate}_{\mathsf{S}'}\ (\mathsf{return_S}\ v), \Sigma \rangle \rightsquigarrow \langle \mathsf{return}_{\mathsf{S}'}\ v, \Sigma \rangle}\ \text{(STM-ElevateRet)}$$

$$\frac{\langle v, s :: \Sigma \rangle \rightsquigarrow \langle t, s' :: \Sigma' \rangle}{\langle \mathsf{runSt}\ v\ s, \Sigma \rangle \rightsquigarrow \langle \mathsf{runSt}\ t\ s', \Sigma' \rangle}\ \text{(STM-RunSt)} \qquad \frac{}{\langle \mathsf{runSt}\ (\mathsf{return}_{(\mathsf{StT}\ \ell\ \tau\ \mathsf{S})}\ v)\ v', \Sigma \rangle \rightsquigarrow \langle \mathsf{return_S}\ \langle v, v' \rangle, \Sigma \rangle}\ \text{(STM-RunStRet)}$$

$$\frac{\langle v, \Sigma \rangle \rightsquigarrow \langle t, \Sigma' \rangle}{\langle \mathsf{runRe}_\tau\ v, \Sigma \rangle \rightsquigarrow \langle \mathsf{runRe}_\tau\ t, \Sigma' \rangle}\ \text{(STM-RunRe)} \qquad \frac{}{\langle \mathsf{runRe}_{\tau''}\ (\mathsf{pause}_{(\mathsf{ReT}\ \tau\ \tau'\ \mathsf{S})}\ v), \Sigma \rangle \rightsquigarrow \langle v\ \texttt{>>=}\ \lambda x.\mathsf{return}\ (\mathsf{inr}_{\tau''}\ x), \Sigma \rangle}\ \text{(STM-RunRePause)}$$

$$\frac{}{\langle \mathsf{runRe}_{\tau''}\ (\mathsf{return}_{(\mathsf{ReT}\ \tau\ \tau'\ \mathsf{S})}\ v), \Sigma \rangle \rightsquigarrow \langle \mathsf{return}_S\ (\mathsf{inl}_{(\tau \to (\tau' \times (\mathsf{ReT}\ \tau\ \tau'\ \mathsf{S}\ \tau'')))}\ v), \Sigma \rangle}\ \text{(STM-RunReRet)}$$

$$\frac{}{\langle \mathsf{unfold}\ v\ v', \Sigma \rangle \rightsquigarrow \left\langle \mathsf{lift}\ (v'\ v)\ \texttt{>>=}\ \lambda u.\begin{pmatrix} \mathsf{case}\ u & (\lambda w.\ \mathsf{return}\ w) \\ & (\lambda w.\mathsf{proj}\ w\ (\lambda x.\lambda y.\ \mathsf{pause}(\mathsf{return}\ \langle x, \lambda z.\ \mathsf{unfold}\ (y\ z)\ v' \rangle))) \end{pmatrix}, \Sigma \right\rangle}\ \text{(STM-Unfold)}$$

$$\frac{}{\langle (\mathsf{pause}\ v)\ \texttt{>>=}\ v', \Sigma \rangle \rightsquigarrow \langle \mathsf{pause}\ (v\ \texttt{>>=}\ \lambda w.\ (\mathsf{proj}\ w\ (\lambda x.\lambda y.\ \mathsf{return}\ \langle x, \lambda z.(y\ z)\ \texttt{>>=}\ v' \rangle)))\ ), \Sigma \rangle}\ \text{(STM-PauseBind)}$$

Figure 3.8: Evaluation rules for monadic reduction. For the sake of readability, type annotations in STM-Unfold and STM-PauseBind are elided.

3.5 Metatheory

In this section we discuss the metatheory of RWC, in particular *type safety* (Section 3.5.1), *strong normalization* (Section 3.5.3), and *soundness* of effect labels (Section 3.5.4).

Type systems-based approaches to language-based security (which seem to have originated with Volpano et al. [106]) usually apply type-soundness arguments to demonstrate the correctness of the type system. This soundness argument follows along these lines: well-typed programs (i.e., programs judged secure by the type system) do not misbehave according to a security model (frequently noninterference-based [39]) defined in terms of the language's semantics. The effect system presented in Section 3.4 can make fine-grained distinctions about memory accesses and, therefore, the soundness of the effect system is highly relevant to multi-level security. For example, "no write down" may be expressed as the type, $\mathsf{StT}\ \mathsf{RW}\ H\ (\mathsf{StT}\ \mathsf{R}\ L\ \mathsf{Id})()$, because any computation with this type may read or write to the high H state, but may only read from the low L state. Similarly, "no read up" is expressed by $\varphi_H : \mathsf{StT}\ \mathsf{RW}\ H\ (\mathsf{StT}\ \mathsf{R}\ L\ \mathsf{Id})()$. The type soundness demonstrated in Section 3.5.4 demonstrates the fidelity of RWC types to its operational semantics.

We shall use $\rightsquigarrow^*$ to denote the reflexive, transitive closure of $\rightsquigarrow$. Thus, we have the following properties of $\rightsquigarrow^*$:

Lemma 3.7. *For all terms t, u, v, and stores $\Sigma, \Sigma', \Sigma''$,*

1. *if $u \rightsquigarrow v$, then $u \rightsquigarrow^* v$. Also, if $\langle u, \Sigma\rangle \rightsquigarrow \langle v, \Sigma'\rangle$, then $\langle u, \Sigma\rangle \rightsquigarrow^* \langle v, \Sigma'\rangle$.*
2. *$u \rightsquigarrow^* u$. Also, $\langle u, \Sigma\rangle \rightsquigarrow^* \langle u, \Sigma\rangle$.*
3. *if $t \rightsquigarrow^* u$ and $u \rightsquigarrow^* v$, then $t \rightsquigarrow^* v$. Also, if $\langle t, \Sigma\rangle \rightsquigarrow^* \langle u, \Sigma'\rangle$ and $\langle u, \Sigma'\rangle \rightsquigarrow^* \langle v, \Sigma''\rangle$, then $\langle t, \Sigma\rangle \rightsquigarrow^* \langle v, \Sigma''\rangle$.*

Moreover, for each single-step reduction rule defined in Figures 3.7 and 3.8 from Section 3.4.3, there exists a corresponding version with $\rightsquigarrow^*$ in place of $\rightsquigarrow$.

3.5.1 Type Safety

As is standard in operational semantics, we take type safety to be the conjunction of *progress*, meaning that any well-typed term (resp. configuration) that is not a value (resp. is not "done") always reduces to something (Theorem 3.8), and *preservation*, meaning that reduction preserves the types of terms (and configurations) (Theorem 3.9).

Theorem 3.8 (Progress)**.** *If* $\{\} \vdash t : \tau$*, then either* t *is a value or there exists* t' *such that* $t \rightsquigarrow t'$*. Also, if* $\langle t, \Sigma\rangle \triangleright M\ \tau$*, then either* $\langle t, \Sigma\rangle$ *is done, or there exist* t' *and* Σ' *such that* $\langle t, \Sigma\rangle \rightsquigarrow \langle t', \Sigma'\rangle$*.*

Theorem 3.9 (Preservation)**.** *If* $\{\} \vdash t : \tau$ *and* $t \rightsquigarrow t'$*, then* $\{\} \vdash t' : \tau$*. Also, if* $\langle t, \Sigma\rangle \triangleright M\ \tau$ *and* $\langle t, \Sigma\rangle \rightsquigarrow \langle t', \Sigma'\rangle$*, then* $\langle t', \Sigma'\rangle \triangleright M\ \tau$*.*

Together, then, these properties imply that well-typed programs cannot go wrong— i.e., evaluation of well-typed programs never "gets stuck"— as specified in Definition 3.10.

Definition 3.10 (Stuck)**.** A term (resp. configuration) is *stuck* if it is neither a value (resp. done configuration) nor reducible to some other term (resp. configuration).

That is, reduction of well-typed terms (and configurations) will not generate something that is neither a value (resp. done configuration), nor reducible (Corollary 3.11).[3]

Corollary 3.11 (Soundness)**.** *If* $\{\} \vdash t : \tau$ *and* $t \rightsquigarrow^* t'$*, then it is not the case that* t' *is stuck. Also, if* $\langle t, \Sigma\rangle \triangleright M\ \tau$ *and* $\langle t, \Sigma\rangle \rightsquigarrow^* \langle t', \Sigma'\rangle$*, then it is not the case that* $\langle t', \Sigma'\rangle$ *is stuck.*

Perhaps surprisingly, the addition of computational features does not substantially complicate the proof of type safety when compared to similar proofs for pure λ-calculi.

[3] Soundness follows by induction over the reduction steps taken. Then, apply Theorem 3.9 to show that reduced term is well-typed, followed by an application of Theorem 3.8 to show that the term is either a value or further reducible.

3.5.2 Canonical Forms

Proofs of metatheoretic theorems about operational semantics (e.g., the proofs of Theorems 3.8, 3.9 and 3.11) are frequently organized in terms of *canonical forms*—that is, closed, well-typed values. The reason for doing so is simply that it drastically reduces the number of cases to be considered in the proof thereby reducing the verification effort. Our canonical forms come in two varieties—the canonical forms of lambda values and canonical forms for monadic expressions stated in Lemmas 3.12 and 3.13, respectively.

Lemma 3.12. *If* $\{\} \vdash v : \tau$ *and* v *is a value, then*

1. *if* τ *is* $\tau_1 \to \tau_2$*, there exists* $x\,u$*, such that* $v = \lambda x : \tau_1.u$*,*
2. *if* τ *is* $\tau_1 \times \tau_2$*, there exists* $t_1 t_2, v = \langle t_1, t_2 \rangle$*,*
3. *if* τ *is* $\tau_1 \ + \ \tau_2$*, there exists* $t', v = \mathsf{inl}_{\tau_2}\, t'$ *or* $v = \mathsf{inr}_{\tau_1}\, t'$*,*
4. *if* τ *is* $()$*,* $v = ()$*,*

Lemma 3.13. *If* $\langle v, \Sigma \rangle \triangleright \mathsf{M}\ \tau$ *and* $\langle v, \Sigma \rangle$ *is a done configuration, then*

1. *if* M *is* S*, there exists* t'*,* $v = \mathsf{return}_{\mathsf{S}}\, t'$*,*
2. *if* M *is* $\mathsf{ReT}\ \tau'\ \tau''\ \mathsf{S}$*, there exists* t'*,* $v = \mathsf{return}_{(\mathsf{ReT}\tau'\tau''\mathsf{S}\ \tau)}\, t'$ *or* $v = \mathsf{pause}_{(\mathsf{ReT}\tau'\tau''\mathsf{S}\ \tau)}\ t'$*.*

The canonical forms for done configurations—and canonical forms in general—greatly reduce the range of potential cases to consider in proofs. In the case of reactive resumptions, the canonical forms for done configurations reflect a fundamental feature of resumptions—namely, that resumptions consume inputs, producing outputs paired with another resumption.

3.5.3 Strong Normalization

Normalization, generally speaking, is a claim about the set of possible reduction sequences of terms. A reduction relation $\rightsquigarrow$ for a language is *weakly normalizing* if, and only if, for

each term t in the language, there is at least one reduction sequence of terms, $t_0 \rightsquigarrow t_1 \rightsquigarrow \cdots \rightsquigarrow t_{n-1} \rightsquigarrow t_n$, such that $t = t_0$ and t_n is irreducible. A reduction relation $\rightsquigarrow$ is *strongly normalizing* if, and only if, every reduction sequence from term t is a prefix of a reduction sequence ending in an irreducible term. Note that strong normalization implies weak normalization, but not vice versa. Note further that these notions of normalization extend in an obvious way from terms to configurations. The Haskell functional language—or, rather, its notion of reduction—is weakly normalizing, but not strongly normalizing, due to Haskell's default lazy evaluation.

Unlike Haskell, RWC enjoys the property of *strong normalization*. This property is especially important in hardware applications for the reason that hardware cannot be allowed to "loop forever" between clock ticks. The computation time between clock ticks must have a static, finite upper bound—this issue is discussed in detail in the references [82, 83]. Strong normalization also makes defined equality easier to work with, as it eliminates the need to account for equality of diverging computations.

The proof of strong normalization (Theorem 3.17) uses an adaptation of the standard *logical relations* technique [67]. A standard proof using logical relations has two steps. First, define a type-indexed collection of relations over terms. The construction of each relation proceeds inductively by utilizing definitions at "smaller types". The construction of these relations use either one of two approaches—*saturated sets* [99, 98], or *reducibility candidates* [38][4]. Second, establishing that relative to their respective type, every well-typed term respects the relation. In short, given a property $\mathbf{P}$, a *logical relation*, $\mathcal{R}_{\{\mathbf{T} \in \mathcal{T}\}}$ (with respect to $\mathbf{P}$), is a collection of type-indexed relations such that for every $\mathbf{R_T} \in \mathcal{R}_{\{\mathbf{T} \in \mathcal{T}\}}$, every element $t \in \mathbf{R_T}$, either has, or preserves $\mathbf{P}$.

We say that a term t *halts* if and only if there exists a value v (not necessarily distinct from t), such that $t \rightsquigarrow^* v$. In a similar fashion, a configuration $\langle t, \Sigma \rangle$ *halts* if, and only if, there exists a done configuration D such that $\langle t, \Sigma \rangle \rightsquigarrow^* D$. The interaction between

[4]Though similar, saturated sets and reducibility candidates are not the same. See [32] for a detailed comparison.

halting and reduction is characterized by the properties collected in Lemma 3.14, while Lemma 3.15 summarizes properties pertaining to reducibility candidates that were used in the course of proving Theorem 3.17. In the case of strong normalization, halting is the property of interest.

Lemma 3.14. *For all terms u, v and stores Σ, Σ',*

1. *If $u \rightsquigarrow v$, then u halts if and only if v halts.*
2. *If $u \rightsquigarrow^* v$, then u halts if and only if v halts.*
3. *If $\langle u, \Sigma\rangle \rightsquigarrow \langle v, \Sigma'\rangle$, then $\langle u, \Sigma\rangle$ halts if and only if $\langle v, \Sigma'\rangle$ halts.*
4. *If $\langle u, \Sigma\rangle \rightsquigarrow^* \langle v, \Sigma'\rangle$, then $\langle u, \Sigma\rangle$ halts if and only if $\langle v, \Sigma'\rangle$ halts.*

Lemma 3.15. *For all terms u, v and types τ,*

1. *If $u \rightsquigarrow v$ and $\mathbf{R}_\tau(u)$, then $\mathbf{R}_\tau(v)$*
2. *If $u \rightsquigarrow^* v$ and $\mathbf{R}_\tau(u)$, then $\mathbf{R}_\tau(v)$*
3. *If $\{\} \vdash u : \tau$, $u \rightsquigarrow v$ and $\mathbf{R}_\tau(v)$, then $\mathbf{R}_\tau(u)$*
4. *If $\{\} \vdash u : \tau$, $u \rightsquigarrow^* v$ and $\mathbf{R}_\tau(v)$, then $\mathbf{R}_\tau(u)$,*
5. *If $\mathbf{R}_\tau(u)$, then u halts.*

We discuss the details of defining reducibility candidates below.

Lemma 3.16. *Let $v_1, \ldots, v_n$ be values of type $\tau_1, \ldots, \tau_n$, such that $\mathbf{R}_{\tau_i}(v_i)$ for each $i = \{1, \ldots, n\}$. Then, if $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash t : \tau$, then $\mathbf{R}_\tau(t[x_1 := v_1] \ldots [x_n := v_n])$.*

Theorem 3.17 follows from Lemma 3.16 using the empty context—keeping in mind that $\mathbf{R}_\tau(t)$, implies that t halts:

Theorem 3.17 (Strong Normalization)**.** *If $\{\} \vdash t : \tau$, then t halts. Also, if $\langle t, \Sigma\rangle \triangleright M\ \tau$, then $\langle t, \Sigma\rangle$ halts.*

Mechanization

Because resumptions are coinductive, and as such, proving that strong normalization holds for configurations requires the use of coinductive proof principles. This is captured by the definition in Figure 3.10. This allows the $\mathbf{R}$ property to be appropriately applied over reactive resumption computations in a manner that ensures productivity. The use of coinduction and coinductive proof principles has been attributed to David Park [93]. Coquand [27] provided a formalization of coinductive types in type theory using a syntactic guardedness condition. This was implemented in Coq by Giménez [36]. To our knowledge, no other mechanized proofs of strong normalization for computational λ-calculi exist in Coq. There is a proof of strong normalization for Moggi's computational metalanguage in Isabelle/HOL using the nominal package [31].

We formalize $\mathbf{R}$ in Coq using a `Fixpoint` definition in Figure 3.9. The straightforward Inductive definition violates Coq's strict positivity requirement—that Inductive definitions cannot have constructors occurring to the left of an arrow [80]. The core of the definition for ordinary lambda terms is standard. However, the monadic components require explanation.

Note that state-layer monadic configurations must reduce to a $\mathsf{return}_{\mathtt{M}}$ in their respective layers. This amounts to requiring that monadic terms have normal forms. That is to say, this reflects a natural requirement of termination—namely, that monadic-reduction performed with regards to a stateful computation results in a value.

Stores feature prominently in the monadic part of our development. It is only natural, then, that we require that stores satisfy two reducibility conditions relative to their corresponding monadic types. We require that terms contained in stores must be values, and that those terms must be in the reducibility sets of their underlying types. These requirements correspond to `store_all_values` and the fixpoint definition `Rsto` in Figure 3.9. As the name suggests, `Rsto` is simply a reducibility requirement for stores.

We embed the coinductive predicate `along_react` inside the fixpoint definition of

```
Fixpoint R (τ:Ty) (t:tm) {struct τ} : Prop := ∅ ⊢ t ∈ τ ∧ halts t ∧
match τ with
| τ1 → τ2          => ∀ t', R τ1 t' → R τ2 (t,t')
| τ1 × τ2          => ∃ t1 t2, t ⇝* ⟨t1,t2⟩ ∧ value t1 ∧ value t2 ∧ R τ1 t1 ∧ R τ2 t2
| τ1 + τ2          => ∃ t', value t' ∧ ((t ⇝* inl_τ2 t' ∧ R τ1 t') ∨ (t ⇝* inr_τ1 t' ∧ R τ2 t'))
| ()               => True
| SM τ'            => ∀ Σ, Rsto SM Σ → ∃ t' Σ', (t,Σ) ⇝* (return_SM t',Σ') ∧ value t' ∧ R τ' t' ∧ Rsto
      SM Σ'
| ReT τi τo SM τ' => ∀ Σ, Rsto SM Σ → ∃ t' Σ', (t,Σ) ⇝* (t',Σ') ∧ value t' ∧ Rsto SM Σ' ∧
along_react (Rsto SM) (R τi) (R τo) (R τ') (t',Σ')
end
with Rsto SM (Σ:store) {struct SM} : Prop := store_all_values Σ ∧ store_matches_mo Σ SM ∧
match SM with
| Id          => True
| StT ℓ τ SM' => ∃ t Σ', R τ t ∧ Rsto SM' Σ' ∧ Σ = t::Σ'
end.

Inductive store_all_values : store → Prop :=
| sav_empty : store_all_values ()
| sav_cons  : ∀ s Σ, value s → store_all_values Σ → store_all_values (s::Σ).

Inductive store_matches_mo : store → Mo → Prop :=
| matches_mo_id  : store_matches_mo () Id
| matches_mo_stt : ∀ SM E τ t Σ, ∅ ⊢ t ∈ τ → store_matches_mo Σ SM → store_matches_mo (t::Σ) (StT E τ
      SM)
| matches_mo_ret : ∀ SM τi τo Σ, store_matches_mo Σ SM → store_matches_mo Σ (ReT τi τo SM).
```

Figure 3.9: The fixpoint definition of logical relation $\mathbf{R}$. The store_matches_mo formalizes the *matches* relation from Fig. 3.6 in Coq.

```
CoInductive along_react : (store → Prop) → (tm → Prop) → (tm → Prop) → (tm → Prop) → configuration
      → Prop :=
| along_return : ∀ (PS:store → Prop) (PI PO PR:tm → Prop) τi τo SM t Σ t' Σ',
(t,Σ) ⇝* (return_(ReT τi τo SM) t',Σ') → value t' → PR t' → PS Σ' → along_react PS PI PO PR (t,Σ)
| along_pause  : ∀ (PS : store → Prop) (PI PO PR : tm → Prop) τi τo SM τ t Σ t1 Σ' vl vr Σ'',
(t,Σ)  ⇝* (pause_(ReT τi τo SM τ) t1,Σ')                                  →
(t1,Σ') ⇝* (return_SM ⟨vl,vr⟩,Σ'')                                        →
value t1 ∧ value vl ∧ value vr                                           →
PS Σ' ∧ PS Σ'' ∧ PO vl                                                    →
(∀ t1, PI t2 → halts (vr t2))                                             →
(∀ t2 Σ'', PI t2 → PS Σ'' → along_react PS PI PO PR ((vr t2),Σ'')) →
along_react PS PI PO PR (t,Σ).
```

Figure 3.10: The Coinductive Predicate along_react

$\mathbf{R}$ as a condition on terms typed in the reactive layer. An added difficulty is that `along_react` needs to be defined lexically prior to the definition of $\mathbf{R}$. As such, `along_react` mentions neither $\mathbf{R}$, nor `Rsto`. Instead, we must use partial application—i.e., $(\mathbf{R}\,\tau)$. These technicalities notwithstanding, the structure of the constructors for `along_react` is fairly straightforward – involving routine reasoning for coinduction.

3.5.4 Soundness of Effect Labels

Since effect labels are meant to track effects and their potential propagation, soundness of effect labels (roughly) corresponds to preservation of security levels indicated by the label, and that stores track such features accordingly. Thus, given well-typed configurations,

$$\frac{}{\mathsf{nil} \overset{\mathsf{Id}(\not{\mathrm{W}})}{=} \mathsf{nil}}\ (\textsc{swnw-id}) \qquad \frac{\Sigma \overset{S(\not{\mathrm{W}})}{=} \Sigma'}{\Sigma \overset{\mathsf{ReT}\ \tau\ \tau'\ S(\not{\mathrm{W}})}{=} \Sigma'}\ (\textsc{swmw-re})$$

$$\frac{\Sigma \overset{S(\not{\mathrm{W}})}{=} \Sigma'}{(s :: \Sigma) \overset{\mathsf{StT}\ \tau \langle\rangle\ S(\not{\mathrm{W}})}{=} (s :: \Sigma')}\ (\textsc{swnw-n}) \qquad \frac{\Sigma \overset{S(\not{\mathrm{W}})}{=} \Sigma'}{(s :: \Sigma) \overset{\mathsf{StT}\ \tau \langle R \rangle\ S(\not{\mathrm{W}})}{=} (s :: \Sigma')}\ (\textsc{swnw-r})$$

$$\frac{\Sigma \overset{S(\not{\mathrm{W}})}{=} \Sigma'}{(s :: \Sigma) \overset{\mathsf{StT}\ \tau \langle W \rangle\ S(\not{\mathrm{W}})}{=} (s' :: \Sigma')}\ (\textsc{swnw-w})$$

$$\frac{\Sigma \overset{S(\not{\mathrm{W}})}{=} \Sigma'}{(s :: \Sigma) \overset{\mathsf{StT}\ \tau \langle RW \rangle\ S(\not{\mathrm{W}})}{=} (s' :: \Sigma')}\ (\textsc{swnw-rw})$$

Figure 3.11: The 'same where no write' relation.

$$\frac{}{\mathsf{nil} \overset{\mathsf{Id}(\mathrm{R})}{=} \mathsf{nil}}\ (\textsc{swr-id}) \qquad \frac{\Sigma \overset{S(\mathrm{R})}{=} \Sigma'}{\Sigma \overset{\mathsf{ReT}\ \tau\ \tau'\ S(\mathrm{R})}{=} \Sigma'}\ (\textsc{swr-re})$$

$$\frac{\Sigma \overset{S(\mathrm{R})}{=} \Sigma'}{(s :: \Sigma) \overset{\mathsf{StT}\ \tau \langle\rangle\ S(\mathrm{R})}{=} (s' :: \Sigma')}\ (\textsc{swr-n}) \qquad \frac{\Sigma \overset{S(\mathrm{R})}{=} \Sigma'}{(s :: \Sigma) \overset{\mathsf{StT}\ \tau \langle R \rangle\ S(\mathrm{R})}{=} (s :: \Sigma')}\ (\textsc{swr-r})$$

$$\frac{\Sigma \overset{S(\mathrm{R})}{=} \Sigma'}{(s :: \Sigma) \overset{\mathsf{StT}\ \tau \langle W \rangle\ S(\mathrm{R})}{=} (s' :: \Sigma')}\ (\textsc{swr-w})$$

$$\frac{\Sigma \overset{S(\mathrm{R})}{=} \Sigma'}{(s :: \Sigma) \overset{\mathsf{StT}\ \tau \langle RW \rangle\ S(\mathrm{R})}{=} (s :: \Sigma')}\ (\textsc{swr-rw})$$

Figure 3.12: The 'same where read' relation.

establishing soundness of effect labels amounts to verifying that monadic-reduction does not alter stores where no writes are allowed (Theorem 3.19); and moreover, that monadic-reduction does not reveal any changes to stores relative to monads with effect labels where only reads are allowed (Theorem 3.21). To that end, we make use of three relations: "same where no writes", "same where read", and "write consistency", written $\overset{M(\not{\mathrm{W}})}{=}$, $\overset{M(\mathrm{R})}{=}$, and *wc*—and defined in Figure 3.11, Figure 3.12, and Figure 3.13—respectively.

Stores (semantically) correspond to state monad transformers. Given a well-typed configuration, the associated store will contain appropriate elements relative to each layer in the state monad transformer stack. In order to update a store, its state monad must

$$\frac{}{\langle \mathsf{nil}, \mathsf{nil} \rangle\ wc\ \langle \mathsf{nil}, \mathsf{nil} \rangle}\ (\textsc{wc-id})$$

$$\frac{\langle \Sigma_1, \Sigma_2 \rangle\ wc\ \langle \Sigma_1', \Sigma_2' \rangle}{\langle s_1 :: \Sigma_1, s_2 :: \Sigma_2 \rangle wc\ \langle s_1 :: \Sigma_1', s_2 :: \Sigma_2' \rangle}\ (\textsc{wc-unchanged})$$

$$\frac{\langle \Sigma_1, \Sigma_2 \rangle\ wc\ \langle \Sigma_1', \Sigma_2' \rangle}{\langle s_1 :: \Sigma_1, s_2 :: \Sigma_2 \rangle wc\ \langle s :: \Sigma_1', s :: \Sigma_2' \rangle}\ (\textsc{wc-changed})$$

Figure 3.13: The write consistency relation.

contain a write label. Lemma 3.18 contains properties used to prove Theorem 3.19.

Lemma 3.18. *For all stores* $\Sigma, \Sigma', \Sigma''$*, and monads* M, M'

1. *If* Σ *matches* M*, then* $\Sigma \overset{\mathsf{M}(\not\mathsf{W})}{=} \Sigma$*.*
2. *if* $\Sigma \overset{\mathsf{M}(\not\mathsf{W})}{=} \Sigma'$ *and* $\Sigma' \overset{\mathsf{M}(\not\mathsf{W})}{=} \Sigma''$*, then* $\Sigma \overset{\mathsf{M}(\not\mathsf{W})}{=} \Sigma''$*.*
3. *If* M *is less permissive than* M' *and* $\Sigma \overset{\mathsf{M}(\not\mathsf{W})}{=} \Sigma'$*, then* $\Sigma \overset{\mathsf{M}'(\not\mathsf{W})}{=} \Sigma'$*.*

Theorem 3.19 (No Forbidden Updates)**.** *If* $\langle t, \Sigma\rangle \triangleright M\ \tau$*, then* $\langle t, \Sigma\rangle \rightsquigarrow \langle t', \Sigma'\rangle$ *implies* $\Sigma \overset{M(\not\mathsf{W})}{=} \Sigma'$*.*

Similarly, reading from a store takes place only relative to state monads that have a read label. This is reflected in the type judgments for put (resp., get) that require a write (resp., read) label in order to be well-typed. Lemma 3.20 contains properties used to prove Theorem 3.21.

Lemma 3.20. *For all stores* $\Sigma, \Sigma', \Sigma''$*, and monads* M, M'

1. *If* Σ *is the same length as* Σ'*, then* $\langle \Sigma, \Sigma'\rangle$ *wc* $\langle \Sigma, \Sigma'\rangle$*.*
2. *If* $\Sigma \overset{\mathsf{M}(R)}{=} \Sigma'$*, then* Σ *is the same length as* Σ'*.*
3. *If* $\langle s :: \Sigma\rangle \overset{\mathsf{StT}\ \tau\ell\ S(R)}{=} \langle s' :: \Sigma'\rangle$*, then* $\langle s :: \Sigma\rangle \overset{\mathsf{S}(R)}{=} \langle s' :: \Sigma'\rangle$*.*
4. *If* $\ell \leq \mathsf{R}$ *and* $\langle s :: \Sigma\rangle \overset{\mathsf{StT}\ \tau\ell\ S(R)}{=} \langle s' :: \Sigma'\rangle$*, then* $s = s'$*.*
5. *If* M *is less permissive than* M' *and* $\Sigma \overset{\mathsf{M}'(R)}{=} \Sigma'$*, then* $\Sigma \overset{\mathsf{M}(R)}{=} \Sigma'$*.*

Theorem 3.21 (Write Consistency)**.** *Suppose* $\Sigma_1 \overset{M(R)}{=} \Sigma_2$ *and that* $\langle t, \Sigma_1\rangle \triangleright M\ \tau$ *and* $\langle t, \Sigma_2\rangle \triangleright M\ \tau$*. Then if* $\langle t, \Sigma_1\rangle \rightsquigarrow \langle t'_1, \Sigma'_1\rangle$ *and* $\langle t, \Sigma_2\rangle \rightsquigarrow \langle t'_2, \Sigma'_2\rangle$*, it follows that* $t'_1 = t'_2$ *and* $\langle \Sigma_1, \Sigma_2\rangle$ *is write consistent with* $\langle \Sigma'_1, \Sigma'_2\rangle$*.*

The intuition underlying write consistency is that when considering a pair of stores Σ_1 and Σ_2, prior to a reduction and a pair of matching stores Σ'_1 and Σ'_2, after a reduction

it is either the case that the pre-reduction stores do not differ from their corresponding post-reduction stores (i.e. because no write takes place) or are equal to each other (i.e., because the same value was written to both Σ_1 and Σ_2).

The pre-stores, as stated in Theorem 3.21, must satisfy the 'same where read' relation. The type system, because of its effect labels and their ordering, restricts admissible alterations to terms and when such changes can be read from stores. This is particularly useful for equational reasoning involving security properties such as noninterference as shown in Figure 3.14. Theorem 3.21 "says": if Σ_1 and Σ_2 are in the 'same where read' relation, then executing term t in Σ_1 and Σ_2 produces both equal resulting terms, t_1 and t_2, resp., as well as write-consistent pre- and post-stores.

3.6 Type-directed Equational Logic for RWC

The rules provided in Figure 3.14 represent the properties of monads present in RWC. Rules (Left-Unit), (Right-Unit), and (Associativity) are the well-known "monad laws" and Rules (Lift-Return) and (Lift->>=) are the "lifting laws" of Liang [60]. Rules (Put-Put), (Put-Get), and (Get-Get), specify the interaction of stateful operations and are drawn from previous work [45]. The $\leq$ relation on state monads is defined in Figure 3.5.

The equational logic of RWC has both atomic noninterference and clobber formalized as consequences of the RWC semantics in Coq; here, we refer to the last three rules of Figure 3.14. These are particular instances for a two layer state monad of the more general rules found in the Coq script repository. Note that, in its Coq formalization, mask computes the appropriate definition from a monad type term taken as an argument. The exact details of this definition need not concern us here, and the interested reader may consult the repository.

$$\frac{t = t' : \tau \in \Gamma}{\Gamma \vdash t = t' : \tau}\ (\textsc{Axiom}) \quad \frac{\Gamma \vdash t = t' : \tau}{\gamma, \Gamma \vdash t = t' : \tau}\ (\textsc{Weakening})$$

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash t = t : \tau}\ (\textsc{Refl}) \quad \frac{\Gamma \vdash t' = t : \tau}{\Gamma \vdash t = t' : \tau}\ (\textsc{Sym}) \quad \frac{\Gamma \vdash t = t' : \tau \quad \Gamma \vdash t' = t'' : \tau}{\Gamma \vdash t = t'' : \tau}\ (\textsc{Trans})$$

$$\frac{y \notin \textsc{fv}(t)}{\Gamma \vdash \lambda x\!:\!\tau.t = \lambda y\!:\!\tau.t[x \coloneqq y] : \tau \to \tau'}\ (\alpha) \quad \frac{\Gamma \vdash \lambda x\!:\!\tau.t : \tau \to \tau' \quad \Gamma \vdash t' : \tau}{\Gamma \vdash (\lambda x\!:\!\tau.t)t' = t[t' \coloneqq x] : \tau'}\ (\beta)$$

$$\frac{\Gamma \vdash t' : \tau \quad \Gamma \vdash t : \tau \to \mathsf{M}\,\tau'}{\Gamma \vdash (\mathsf{return}_{\mathsf{M}}\, t') \gg\!= t = t\ t' : \mathsf{M}\,\tau'}\ (\textsc{Left-Unit}) \quad \frac{\Gamma \vdash t : \mathsf{M}\,\tau}{\Gamma \vdash t \gg\!= \lambda x\!:\!\tau.(\mathsf{return}_{\mathsf{M}}\, x) = t : \mathsf{M}\,\tau}\ (\textsc{Right-Unit})$$

$$\frac{\Gamma \vdash t : \mathsf{M}\,\tau \quad \Gamma \vdash t' : \tau \to \mathsf{M}\,\tau' \quad \Gamma \vdash t'' : \tau' \to \mathsf{M}\,\tau'' \quad x \notin FV(t')}{\Gamma \vdash (t \gg\!= t') \gg\!= t'' = t \gg\!= (\lambda x\!:\!\tau.t'x \gg\!= t'') : \mathsf{M}\,\tau''}\ (\textsc{associativity-}\gg\!=)$$

$$\frac{\Gamma \vdash \mathsf{return}_{\mathsf{M}}\, t : \mathsf{M}\,\tau}{\Gamma \vdash \mathsf{lift}_{\mathsf{M}'}(\mathsf{return}_{\mathsf{M}}\ t) = \mathsf{return}_{\mathsf{M}'}\ t : \mathsf{M}'\,\tau}\ (\textsc{Lift-Return})$$

$$\frac{\Gamma \vdash t : \mathsf{M}\,\tau \quad \Gamma \vdash t' : \tau \to \mathsf{M}\,\tau' \quad x \notin FV(t')}{\Gamma \vdash \mathsf{lift}_M\,(t \gg\!= t') = (\mathsf{lift}_M\ t) \gg\!= (\lambda(x : \tau).\,\mathsf{lift}_M\ (t'\,x))}\ (\textsc{Lift-}\gg\!=)$$

$$\frac{\Gamma \vdash \mathsf{put}\ t : \mathsf{StT}\,\ell\,\tau\,\mathsf{S}\,() \quad \Gamma \vdash \mathsf{put}\ t' : \mathsf{StT}\,\ell\,\tau\,\mathsf{S}\,()}{\Gamma \vdash (\mathsf{put}\ t \gg \mathsf{put}\ t') = \mathsf{put}\ t' : \mathsf{StT}\,\ell\,\tau\,\mathsf{S}\,()}\ (\textsc{Put-Put})$$

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash (\mathsf{put}\ t \gg \mathsf{get}_{(\mathsf{StT}\,\mathsf{RW}\,\tau\,\mathsf{S})}) = \mathsf{put}\ t \gg \mathsf{return}_{(\mathsf{StT}\,\mathsf{RW}\,\tau\,\mathsf{S})}\ t : \mathsf{StT}\,\mathsf{RW}\,\tau\,\mathsf{S}\,\tau}\ (\textsc{Put-Get})$$

$$\frac{\langle \mathsf{R} \rangle \leq \ell,\ \text{such that } \mathsf{M} = \mathsf{StT}\,\ell\,\tau\,\mathsf{S}}{\Gamma \vdash \mathsf{get}_{\mathsf{M}} \gg\!= \lambda x\!:\!\tau.\,\mathsf{get}_{\mathsf{M}} \gg\!= \lambda y\!:\!\tau.\,\mathsf{return}_{\mathsf{M}}\langle x, y\rangle = \mathsf{get}_{\mathsf{M}} \gg\!= \lambda z\!:\!\tau.\,\mathsf{return}_{\mathsf{M}}\langle z, z\rangle\ :\ \mathsf{M}\ (\tau \times \tau)}\ (\textsc{Get-Get})$$

$$\frac{\Gamma \vdash t\colon \mathsf{S}\,\tau \quad \mathsf{S}\ \text{is}\ \mathsf{StT}\,\mathsf{RW}\,\tau\,(\mathsf{StT}\langle\rangle\,\tau'\,\mathsf{Id})}{\Gamma \vdash t \gg (\mathsf{mask}\,\mathsf{S}) = \mathsf{mask}\,\mathsf{S} : \mathsf{S}\,()}\ (\textsc{Clobber-Lo})$$

$$\frac{\Gamma \vdash \mathsf{lift}_S\,t\colon \mathsf{S}\,() \quad \mathsf{S}\ \text{is}\ \mathsf{StT}\,\langle\rangle\,\tau\,(\mathsf{StT}\,\mathsf{RW}\,\tau'\,\mathsf{Id})}{\Gamma \vdash \mathsf{lift}_S\,t \gg (\mathsf{mask}\,\mathsf{S}) = (\mathsf{mask}\,\mathsf{S}) \gg \mathsf{lift}_S\,t : \mathsf{S}\,()}\ (\textsc{Atomic Noninterference})$$

Figure 3.14: Type-directed Equational Logic for RWC

3.7 Conclusions

This chapter presented a mechanized formal semantics for the functional hardware description language ReWire and, as such, provides a foundation for high assurance hardware design and implementation. The semantics presented here is of the small-step operational variety, which is, at first blush, somewhat surprising. ReWire is a computational λ-calculus in the sense of Moggi [69], and, therefore, possesses a "built-in" denotational semantics based in categorical language semantics [68] which has been discussed elsewhere [82]. But, generally speaking, small-step operational semantics are more readily mechanized in a theorem prover like Coq and this was a primary motivation for pursuing an operational approach.

Synchronous hardware is generally assumed to be non-terminating and that motivates the use of ReWire's core abstraction—potentially infinite resumption-monadic computations—for modeling hardware [83]. Formalizing resumptions in Coq involved technical challenges that required some ingenuity to overcome; these challenges and our approach to overcoming them were discussed in detail in Section 3.5. To the authors' best knowledge, the coinductive style of defining logical relations in Coq is apparently an innovation that may be of use to other researchers in formal methods and interactive theorem proving.

ReWire inherits its purity (i.e., freedom from side effects) from Haskell, and purity, in turn, made the task of formally specifying ReWire relatively straightforward. Were ReWire embedded in an impure functional language (e.g., OCaml or Scala[5]), its resulting semantic specification would have necessarily been more complicated in order to account for the host language's side effects. Any model of synchronous hardware will be complex—but, that being said, the purity of ReWire contributed to simplifying its formalized semantics.

[5]Homepages: `https://ocaml.org` and `https://www.scala-lang.org`, respectively.

Chapter 4

Summary and concluding remarks

The ReWire methodology differs fundamentally from the type-based approach to secure hardware (e.g., that of Caisson [58], Sapper [59], and SecVerilog [114]) in three important respects. Firstly, ReWire is a functional language (a subset of Haskell) and has the benefit, we would argue, of the expressiveness of functional languages. Secondly, ReWire possesses a formal semantics and equational theory mechanized in the Coq theorem proving system, allowing security verification to be automatically checked with the attendant increased assurance. Thirdly, and most importantly, ReWire's type system is not a security type system in the usual sense [88]. Security verification in ReWire is not fully automatic via a security type system, but, rather, the equational style of security verification of our previous work [45, 83] is supported by an effects type system based on the marriage of effects and monads [109]. However, we believe that ReWire's being a pure functional language will support the adaptation of ideas from language-based security to the construction of high assurance, secure hardware via extensions to the ReWire type system.

The ReWire methodology, therefore, occupies a middle ground between the security via typechecking approach of Caisson and SecVerilog and traditional hardware verification with theorem provers [63]. It combines the advantages of both—static checking on the one hand and deductive reasoning on the other—with the expressive power of functional languages. Delite—a compiler framework for parallel embedded domain-specific

languages (EDSLs) targeted to produce hardware—exhibits what its creators call "the three P's" [56]: productivity, performance and portability. Our previous work [43, 42, 83] demonstrates that ReWire possesses what "the three P's" [56] and the current work shows ReWire also possesses a fourth "P": provability. Follow-on articles will present the formalizations of previously published verifications of ReWire devices [43, 83].

The CompCert [57] project mechanizes both a source language's semantics and compiler in Coq, thereby providing the foundation for (1) verifying properties of C source programs and (2) compiling those programs to efficient implementations in a verifiably property-preserving manner. One particular strength of the CompCert approach is that other tools may be mechanized in Coq as well (e.g., static analysis tools, etc., from the Verified Software Toolchain [107]) to provide increased automation and trust to the whole workflow. The current work is motivated by the goal of producing trusted hardware in the same manner as CompCert supports trusted C implementations. This is, admittedly, a very ambitious goal, but the current work is an early, yet important, step in this program. The current work also provides an important first step towards the formal verification of the ReWire compiler.

BIBLIOGRAPHY

[1] A. Abel. “A polymorphic lambda-calculus with sized higher-order types”. PhD thesis. Ludwig-Maximilians-Universität München, 2006.

[2] A. Abel. “Semi-continuous Sized Types and Termination”. In: *Logical Methods in Computer Science* Volume 4, Issue 2 (Apr. 2008).

[3] D. Andrews. *Will the Future Success of Reconfigurable Computing Require a Paradigm Shift in Our Research Community's Thinking?* Keynote address, Applied Reconfigurable Computing. http://hthreads.csce.uark.edu/mediawiki/images/d/d8/Arc-presentation.pdf. 2015.

[4] A. Azevedo de Amorim et al. “A Verified Information-flow Architecture”. In: *POPL*. 2014, pp. 165–178.

[5] C. Baaij and J. Kuper. “Using Rewriting to Synthesize Functional Languages to Digital Circuits”. In: *Trends in Fun. Prog.* Vol. 8322. LNCS. 2014, pp. 17–33.

[6] J. Bachrach et al. “Chisel: constructing hardware in a Scala embedded language”. In: *DAC*. 2012, pp. 1216–1225.

[7] D. Bacon, R. Rabbah, and S. Shukla. “FPGA Programming for the Masses”. In: *Queue* 11.2 (Feb. 2013), 40:40–40:52.

[8] H. Barendregt. “Functional Programming and Lambda Calculus”. In: *Formal Models and Semantics*. Ed. by J. V. Leeuwen. Vol. B. Handbook of Theoretical Computer Science. Amsterdam: Elsevier, 1990. Chap. 7, pp. 321–363.

[9] Barthe, G. et al. “Type-based termination of recursive definitions”. In: *Mathematical Structures in Computer Science* 14.1 (2004), pp. 97–141.

[10] G. Barthe, B. Grégoire, and C. Riba. “Type-Based Termination with Sized Products”. In: *Computer Science Logic*. Ed. by M. Kaminski and S. Martini. Vol. 5213. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 493–507.

[11] L. Baugh, N. Neelakantam, and C. Zilles. “Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory”. In: *Proceedings of the 35th Annual International Symposium on Computer Architecture*. ISCA ’08. 2008, pp. 115–126.

[12] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

[13] P. Bjesse et al. “Lava: Hardware design in Haskell”. In: *3rd ICFP*. 1998, pp. 174–184.

[14] B. Homepage. http://bluespec.com. July 2017.

[15] T. Braibant and A. Chlipala. “Formal Verification of Hardware Synthesis”. In: *CAV*. 2013, pp. 213–228.

[16] G. Cabodi and M. Murciano. “BDD-Based Hardware Verification”. In: *6th Inter. Conf. on Formal Methods for the Design of Computer, Communication, and Software Systems*. SFM’06. 2006, pp. 78–107.

[17] J. Choi et al. “Kami: A Platform for High-level Parametric Hardware Specification and Its Modular Verification”. In: *Proc. ACM Program. Lang.* 1.ICFP (Aug. 2017), 24:1–24:30.

[18] A. Church. “A Formulation of the Simple Theory of Types”. In: *Journal of Symbolic Logic* 5 (1940), pp. 56–68.

[19] A. Church. “A Note on the Entscheidungsproblem”. In: *Journal of Symbolic Logic* 1 (1936), pp. 40–41.

[20] A. Church. *The Calculi of Lambda-Conversion*. Annals of Mathematics Studies. Princeton University Press, 1941.

[21] K. Claessen and J. Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *SIGPLAN Not.* 35.9 (Sept. 2000), pp. 268–279.

[22] D. Cock, G. Klein, and T. Sewell. "Secure Microkernels, State Monads and Scalable Refinement". In: *TPHOLs*. 2008, pp. 167–182.

[23] *Prover9 and Mace4*. https://www.cs.unm.edu/~mccune/mace4/.

[24] *Automath*. http://www.cs.ru.nl/~freek/aut/.

[25] *Isabelle*. https://isabelle.in.tum.de/.

[26] *The Coq Proof Assistant*. https://coq.inria.fr.

[27] T. Coquand. "Infinite objects in type theory". In: *Types for Proofs and Programs: International Workshop TYPES'93 Nijmegen, The Netherlands, May 24–28, 1993 Selected Papers*. Ed. by H. Barendregt and T. Nipkow. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 62–78.

[28] K. Crary, A. Kliger, and F. Pfenning. "A monadic analysis of information flow security with mutable state". In: *JFP* 15.2 (Mar. 2005), pp. 249–291.

[29] Curry, Haskell B. and Feys, R. *Combinatory Logic*. Vol. I. Studies in Logic and the Foundations of Mathematics. North-Holland Publishing Company, 1958.

[30] Curry, Haskell B., Hindley, J. R., and Seldin, J. P. *Combinatory Logic*. Vol. II. Studies in Logic and the Foundations of Mathematics. North-Holland Publishing Company, 1972.

[31] C. Doczkal and J. Schwinghammer. "Formalizing a Strong Normalization Proof for Moggi's Computational Metalanguage: A Case Study in Isabelle/HOL-nominal". In: *Proceedings of the Fourth International Workshop on Logical Frameworks and*

Meta-Languages: Theory and Practice. LFMTP '09. Montreal, Quebec, Canada: ACM, 2009, pp. 57–63.

[32] J. H. Gallier. "On Girard's "Candidates de Reducibilite"". In: *Logic and Computer Science*. Academic Press, 1990, pp. 123–204.

[33] P. Gammie. "Synchronous Digital Circuits As Functional Programs". In: *ACM Comput. Surv.* 46.2 (Nov. 2013), 21:1–21:27.

[34] N. George et al. "Hardware system synthesis from Domain-Specific Languages". In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. Sept. 2014, pp. 1–8.

[35] D. Ghica and A. Jung. "Categorical semantics of digital circuits". In: *FMCAD*. 2016.

[36] E. Giménez. "Codifying guarded definitions with recursive schemes". In: *Types for Proofs and Programs: International Workshop TYPES '94 Båstad, Sweden, June 6–10, 1994 Selected Papers*. Ed. by P. Dybjer, B. Nordström, and J. Smith. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 39–59.

[37] E. Giménez. "Structural recursive definitions in type theory". In: *Automata, Languages and Programming*. Ed. by K. G. Larsen, S. Skyum, and G. Winskel. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 397–408.

[38] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and types*. Vol. 7. Cambridge University Press Cambridge, 1989.

[39] J. Goguen and J. Meseguer. "Unwinding and Inference Control". In: *IEEE Symp. on Security and Privacy*. 1984, pp. 75–86.

[40] S. Goncharov and L. Schröder. "A coinductive calculus for asynchronous side-effecting processes". In: *Proc. of the 18th International Conf. on Fundamentals of Computation Theory*. 2011, pp. 276–287.

[41] M. Gordon. “The semantic challenge of Verilog HDL”. In: *Logic in Computer Science, 1995. LICS ’95. Proceedings., Tenth Annual IEEE Symposium on*. June 1995, pp. 136–145.

[42] I. Graves et al. “Hardware Synthesis from Functional Embedded Domain-Specific Languages: A Case Study in Regular Expression Compilation”. In: *Applied Reconfigurable Computing*. Vol. 9040. LNCS. 2015, pp. 41–52.

[43] I. Graves et al. “Provably Correct Development of Reconfigurable Hardware Designs via Equational Reasoning”. In: *IEEE Inter. Conf. on Field-Programmable Technology (ICFPT)*. 2015, pp. 160–171.

[44] W. Harrison. “The Essence of Multitasking”. In: *Algebraic Methodology and Software Technology*. 2006, pp. 158–172.

[45] W. Harrison and J. Hook. “Achieving information flow security through monadic control of effects”. In: *JCS* 17 (5 Oct. 2009), pp. 599–653.

[46] W. Harrison, A. Procter, and G. Allwein. “Model-driven Design & Synthesis of the SHA-256 Cryptographic Hash Function in ReWire”. In: *Proceedings of the 27th International Symposium on Rapid System Prototyping (RSP)*. 2016, pp. 1–7.

[47] W. Harrison et al. “A Programming Model for Reconfigurable Computing Based in Functional Concurrency”. In: *11th Inter. Symp. on Reconfigurable Communication-centric Systems-on-Chip*. 2016.

[48] G. J. Holzmann. “The Model Checker SPIN”. In: *IEEE Transactions on Software Engineering* 23.5 (May 1997), pp. 279–295.

[49] B. Huffman. “HOLCF ’11: A Definitional Domain Theory for Verifying Functional Programs”. PhD thesis. Portland State University, 2012.

[50] T. Huffmire et al. “Enforcing memory policy specifications in reconfigurable hardware”. In: *Computers & Security* 27.5–6 (2008), pp. 197–215.

[51] T. Huffmire et al. *Handbook of FPGA Design Security*. Springer, 2010.

[52] T. Huffmire et al. "Policy-Driven Memory Protection for Reconfigurable Hardware". In: *ESORICS*. Vol. 4189. LNCS. 2006, pp. 461–478.

[53] J. Hughes, L. Pareto, and A. Sabry. "Proving the Correctness of Reactive Systems Using Sized Types". In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '96. St. Petersburg Beach, Florida, USA: ACM, 1996, pp. 410–423.

[54] C. Kloos and P. Breuer, eds. *Formal Semantics for VHDL*. Kluwer Academic Publishers, 1995.

[55] L. Lamport and P. M. Melliar-Smith. "Synchronizing Clocks in the Presence of Faults". In: *Journal of the Association of Computing Machinery* 32.1 (Jan. 1985), pp. 52–78.

[56] H. Lee et al. "Implementing Domain-Specific Languages for Heterogeneous Parallel Computing". In: *IEEE Micro* 31.5 (Sept. 2011), pp. 42–53.

[57] X. Leroy. "Formal Verification of a Realistic Compiler". In: *Commun. ACM* 52.7 (July 2009), pp. 107–115.

[58] X. Li et al. "Caisson: a hardware description language for secure information flow". In: *PLDI*. 2011, pp. 109–120.

[59] X. Li et al. "Sapper: A Language for Hardware-level Security Policy Enforcement". In: *ASPLOS*. 2014.

[60] S. Liang, P. Hudak, and M. Jones. "Monad Transformers and Modular Interpreters". In: *POPL*. 1995, pp. 333–343.

[61] K. L. McMillan. "The SMV System". In: *Symbolic Model Checking*. Boston, MA: Springer US, 1993. Chap. 4, pp. 61–85.

[62] A. Megacz. “Hardware Design with Generalized Arrows”. In: *Proceedings of the 23rd International Conference on Implementation and Application of Functional Languages*. IFL’11. Lawrence, KS: Springer-Verlag, 2012, pp. 164–180.

[63] T. Melham. *Higher Order Logic and Hardware Verification*. Vol. 31. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1993.

[64] C. repository for MEMOCODE. https://goo.gl/FYf6xU. July 2017.

[65] N. P. Mendler. “Inductive types and type constraints in the second-order lambda calculus”. In: *Annals of pure and Applied logic* 51.1-2 (1991), pp. 159–172.

[66] R. Milner. “A theory of type polymorphism in programming”. In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348–375.

[67] J. Mitchell. *Foundations for Programming Languages*. MIT Press Cambridge, 1996.

[68] E. Moggi. *An Abstract View of Programming Languages*. Tech. rep. ECS-LFCS-90-113. Department of Computer Science, Edinburgh University, 1990.

[69] E. Moggi. “Notions of computation and monads”. In: *Information and Computation* 93.1 (July 1991), pp. 55–92.

[70] L. de Moura et al. “SAL 2”. In: *Computer Aided Verification*. Ed. by R. Alur and D. A. Peled. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 496–500.

[71] A. Myers. Personal communication. Mar. 7, 2017.

[72] A. Nanevski et al. “Ynot: Dependent Types for Imperative Programs”. In: *ICFP*. 2008, pp. 229–240.

[73] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. 1999.

[74] R. S. Nikhil and Arvind. “What is Bluespec?” In: *SIGDA Newsl.* 39.1 (Jan. 2009), pp. 1–1.

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

[76] S. Ouchani, O. A. Mohamed, and M. Debbabi. “A formal verification framework for Bluespec System Verilog”. In: *Proceedings of the 2013 Forum on specification and Design Languages (FDL)*. Sept. 2013, pp. 1–7.

[77] L. Pareto. “Types for Crash Prevention”. PhD thesis. Chalmers University of Technology, 2000.

[78] S. Peyton Jones, ed. *Haskell 98 Language and Libraries, the Revised Report*. Cambridge University Press, 2003, p. 272.

[79] Pierce, Benjamin C. *Types and Programming Languages*. MIT Press, 2002.

[80] Pierce, Benjamin C. et al. *Software Foundations*. 2010.

[81] Plotkin, Gordon D. *A structural approach to operational semantics*. Tech. rep. Aarhus University, 1981.

[82] A. Procter. “Semantics-Driven Design and Implementation of High-Assurance Hardware”. PhD thesis. University of Missouri, 2014. Department of Computer Science., 2014.

[83] A. Procter et al. “A Principled Approach to Secure Multi-core Processor Design with ReWire”. In: *ACM TECS* 16.2 (Feb. 2017), 33:1–33:25.

[84] A. Procter et al. “Semantics Driven Hardware Design, Implementation, and Verification with ReWire”. In: *ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES)*. 2015.

[85] T. N. Reynolds et al. “A Core Calculus for Secure Hardware: Its Formal Semantics and Proof System”. In: *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE17)*. 2017.

[86] T. N. Reynolds et al. “The Mechanized Marriage of Effects and Monads with Applications to High-assurance Hardware”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 18.1 (Jan. 2019), 6:1–6:26.

[87] D. Richards and D. Lester. “A monadic approach to automated reasoning for Bluespec SystemVerilog”. In: *Innovations in Systems and Software Engineering* 7.2 (Mar. 2011), p. 85.

[88] A. Sabelfeld and A. Myers. “Language-based Information-flow Security”. In: *IEEE Journ. on Sel. Areas in Commun.* 21.1 (Jan. 2003).

[89] J. Sacchini. “On type-based termination and dependent pattern matching in the calculus of inductive constructions”. PhD thesis. École Nationale Supérieure des Mines de Paris, 2011.

[90] J. L. Sacchini. “Linear Sized Types in the Calculus of Constructions”. In: *Functional and Logic Programming*. Ed. by M. Codish and E. Sumii. Cham: Springer International Publishing, 2014, pp. 169–185.

[91] I. Sander and A. Jantsch. “Modelling Adaptive Systems in ForSyDe”. In: *Electronic Notes in Theoretical Computer Science* 200.2 (2008), pp. 39–54.

[92] I. Sander and A. Jantsch. “System modeling and transformational design refinement in ForSyDe”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 23.1 (2004), pp. 17–32.

[93] D. Sangiorgi. “On the Origins of Bisimulation and Coinduction”. In: *ACM Trans. Program. Lang. Syst.* 31.4 (May 2009), 15:1–15:41.

[94] L. Schröder and T. Mossakowski. “HasCasl: Integrated higher-order specification and program development”. In: *Theoretical Computer Science* 410.12 (2009), pp. 1217–1260.

[95] M. Sheeran. “muFP, a Language for VLSI Design”. In: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. LFP ’84. Austin, Texas, USA: ACM, 1984, pp. 104–112.

[96] G. E. Suh et al. “Secure Program Execution via Dynamic Information Flow Tracking”. In: *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XI. Boston, MA, USA: ACM, 2004, pp. 85–96.

[97] W. Swierstra. “A Hoare Logic for the State Monad”. In: *TPHOLs*. 2009, pp. 440–451.

[98] W. W. Tait. “A Realizability Interpretation of the Theory of Species”. In: *Logic Colloquium*. Ed. by R. Parikh. Vol. 453. Lectures Notes in Mathematics. Boston: Springer-Verlag, 1975, pp. 240–251.

[99] W. W. Tait. “Intensional interpretations of functionals of finite type I”. In: *The journal of symbolic logic* 32.2 (1967), pp. 198–212.

[100] M. Tehranipoor and C. Wang. *Introduction to Hardware Security and Trust*. Springer Publishing Company, Incorporated, 2011.

[101] M. Tiwari et al. “Complete Information Flow Tracking from the Gates Up”. In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIV. Washington, DC, USA: ACM, 2009, pp. 109–120.

[102] M. Tiwari et al. “Complete Information Flow Tracking from the Gates Up”. In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIV. Washington, DC, USA: ACM, 2009, pp. 109–120.

[103] M. Tiwari et al. “Crafting a Usable Microkernel, Processor, and I/O System with Strict and Provable Information Flow Security”. In: *Proceedings of the 38th Annual ISCA*. 2011, pp. 189–200.

[104] M. Tiwari et al. “Execution leases: A hardware-supported mechanism for enforcing strong non-interference”. In: *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. Dec. 2009, pp. 493–504.

[105] S. M. Trimberger and J. J. Moore. “FPGA Security: Motivations, Features, and Applications”. In: *Proceedings of the IEEE* 102.8 (Aug. 2014), pp. 1248–1265.

[106] D. Volpano, C. Irvine, and G. Smith. “A Sound Type System for Secure Flow Analysis”. In: *J. Comput. Secur.* 4.2-3 (Jan. 1996), pp. 167–187.

[107] *Verified Software Toolchain*. http://vst.cs.princeton.edu.

[108] A. P. W. Harrison and G. Allwein. “The Confinement Problem in the Presence of Faults”. In: *ICFEM*. 2012, pp. 182–197.

[109] P. Wadler. “The Marriage of Effects and Monads”. In: *ICFP*. 1998, pp. 63–74.

[110] P. Weis and X. Leroy. *Le langage Caml*. 2nd ed. Dunod, 1999. 370 pp.

[111] H. Xi. “Dependent Types for Program Termination Verification”. In: *Higher Order Symbolic Computation* 15.1 (Mar. 2002), pp. 91–131.

[112] N. Zeldovich et al. “Hardware Enforcement of Application Security Policies Using Tagged Memory”. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI’08. San Diego, California: USENIX Association, 2008, pp. 225–240.

[113] K. Zhai et al. “Hardware Synthesis from a Recursive Functional Language”. In: *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*. CODES ’15. Amsterdam, The Netherlands: IEEE Press, 2015, pp. 83–93.

[114] D. Zhang et al. *A Hardware Design Language for Efficient Control of Timing Channels*. Tech. rep. 2014-04-10. Extended version of the authors’ ASPLOS15 paper. Dept. of Computer Science, Cornell University, 2014.

Appendix A

BTC COQ Code

A.1 Syntax

A.1.1 Types

`Inductive` *Ty* : `Type` :=

| *TArrow* : *nat* → *Ty* → *Ty* → *Ty*

| *TProd* : *Ty* → *Ty* → *Ty*

| *TSum* : *Ty* → *Ty* → *Ty*

| () : *Ty*.

A.1.2 Terms

`Inductive` *term* : `Type` :=

| var :> *id* → *term*

| *tapp* : *term* → *term* → *term*

| λ : *id* → *Ty* → *term* → *term*

| `nil` : *term*

| *tpair* : *term* → *term* → *term*

| π_1 : *term* → *term*

| π_2 : *term* → *term*

| inl : *term* → *Ty* → *term*

| inr : *term* → *Ty* → *term*

| $case$: *term* → *term* → *term* → *term* .

A.1.3 Values

```
Inductive value : term → Prop :=
  | v_abs : ∀ x T t,
      value (λ x T t)
  | v_unit : value nil
  | v_pair : ∀ t1 t2,
      value t1 → value t2 → value (tpair t1 t2)
  | v_inl : ∀ t T,
      value t → value (inl t T)
  | v_inr : ∀ t T,
      value t → value (inr t T).
```

A.2 Typing Judgments

A.2.1 For terms

```
Definition context := partial_map Ty.

Inductive has_type : context → term → Ty → nat → Prop :=
  | T_Var : ∀ Γ x T,
                Γ x = Some T →
                has_type Γ (var x) T O
  | T_Abs : ∀ Γ x T1 T2 t n,
                has_type (extend Γ x T1) t T2 n →
                has_type Γ (λ x T1 t) (TArrow n T1 T2) O
  | T_App : ∀ T1 T2 Γ f t1 n m p,
```

$$\mathit{has\_type}\ \Gamma\ f\ (\mathit{TArrow}\ n\ T_1\ T_2)\ m \rightarrow$$

$$\mathit{has\_type}\ \Gamma\ t_1\ T_1\ p \rightarrow$$

$$\mathit{has\_type}\ \Gamma\ (\mathit{tapp}\ f\ t_1)\ T_2\ (n + m + p + 1)$$

| *T_Nil* : $\forall\ \Gamma$,

$$\mathit{has\_type}\ \Gamma\ \texttt{nil}\ ()\ O$$

| *T_Pair* : $\forall\ \Gamma\ T_1\ T_2\ t_1\ t_2\ n\ m$,

$$\mathit{has\_type}\ \Gamma\ t_1\ T_1\ n \rightarrow$$

$$\mathit{has\_type}\ \Gamma\ t_2\ T_2\ m \rightarrow$$

$$\mathit{has\_type}\ \Gamma\ (\mathit{tpair}\ t_1\ t_2)\ (\mathit{TProd}\ T_1\ T_2)\ (n + m)$$

| *T_Pi1* : $\forall\ \Gamma\ T_1\ T_2\ t\ n$,

$$\mathit{has\_type}\ \Gamma\ t\ (\mathit{TProd}\ T_1\ T_2)\ n \rightarrow$$

$$\mathit{has\_type}\ \Gamma\ (\pi_1\ t)\ T_1\ (n + 1)$$

| *T_Pi2* : $\forall\ \Gamma\ T_1\ T_2\ t\ n$,

$$\mathit{has\_type}\ \Gamma\ t\ (\mathit{TProd}\ T_1\ T_2)\ n \rightarrow$$

$$\mathit{has\_type}\ \Gamma\ (\pi_2\ t)\ T_2\ (n + 1)$$

| *T_Inl* : $\forall\ \Gamma\ T_1\ T_2\ t\ n$,

$$\mathit{has\_type}\ \Gamma\ t\ T_1\ n \rightarrow$$

$$\mathit{has\_type}\ \Gamma\ (inl\ t\ T_2)\ (\mathit{TSum}\ T_1\ T_2)\ n$$

| *T_Inr* : $\forall\ \Gamma\ T_1\ T_2\ t\ n$,

$$\mathit{has\_type}\ \Gamma\ t\ T_2\ n \rightarrow$$

$$\mathit{has\_type}\ \Gamma\ (inr\ t\ T_1)\ (\mathit{TSum}\ T_1\ T_2)\ n$$

| *T_Case* : $\forall\ \Gamma\ T_l\ T_r\ \mathit{Tres}\ t_s\ t_l\ t_r\ n_l\ n_r\ n\ m\ p$,

$$\mathit{has\_type}\ \Gamma\ t_s\ (\mathit{TSum}\ T_l\ T_r)\ n \rightarrow$$

$$\mathit{has\_type}\ \Gamma\ t_l\ (\mathit{TArrow}\ n_l\ T_l\ \mathit{Tres})\ m \rightarrow$$

$$\mathit{has\_type}\ \Gamma\ t_r\ (\mathit{TArrow}\ n_r\ T_r\ \mathit{Tres})\ p \rightarrow$$

$$\mathit{has\_type}\ \Gamma\ (case\ t_s\ t_l\ t_r)\ \mathit{Tres}\ (n + (\max\ (n_l + m)\ (n_r + p)) + 2).$$

A.3 Substitution

```
Reserved Notation "'[' x ':=' s ']' t" (at level 20).

Fixpoint subst (x:id) (s:term) (t:term) : term :=
  match t with
  | var x' ⇒
      if eq_id_dec x x' then s else t
  | tapp t1 t2 ⇒
      tapp ([x:=s] t1) ([x:=s] t2)
  | λ x' T t1 ⇒
      λ x' T (if eq_id_dec x x' then t1 else ([x:=s] t1))
  | nil ⇒
      nil
  | tpair t1 t2 ⇒
      tpair ([x:=s] t1) ([x:=s] t2)
  | π1 t1 ⇒
      π1 ([x:=s] t1)
  | π2 t1 ⇒
      π2 ([x:=s] t1)
  | inl t1 T ⇒
      inl ([x:=s] t1) T
  | inr t1 T ⇒
      inr ([x:=s] t1) T
  | case t1 t2 t3 ⇒
      case ([x:=s] t1) ([x:=s] t2) ([x:=s] t3)
  end
where "'[' x ':=' s ']' t" := (subst x s t).
```

A variable x appears free in a term t .

```
Inductive appears_free_in : id → term → Prop :=
  | afi_var : ∀ x,
      appears_free_in x (var x)
  | afi_app1 : ∀ x t1 t2,
      appears_free_in x t1 →
      appears_free_in x (tapp t1 t2)
  | afi_app2 : ∀ x t1 t2,
      appears_free_in x t2 →
      appears_free_in x (tapp t1 t2)
  | afi_abs : ∀ x y T11 t12,
      y ≠ x →
      appears_free_in x t12 →
      appears_free_in x (λ y T11 t12)
  | afi_pair1 : ∀ x t1 t2,
      appears_free_in x t1 →
      appears_free_in x (tpair t1 t2)
  | afi_pair2 : ∀ x t1 t2,
      appears_free_in x t2 →
      appears_free_in x (tpair t1 t2)
  | afi_pi1 : ∀ x t,
      appears_free_in x t →
      appears_free_in x (π1 t)
  | afi_pi2 : ∀ x t,
      appears_free_in x t →
      appears_free_in x (π2 t)
  | afi_tinl : ∀ x t TR,
```

appears_free_in x t →

appears_free_in x (inl *t TR*)

| *afi_tinr* : ∀ *x t TL*,

appears_free_in x t →

appears_free_in x (inr *t TL*)

| *afi_tcase1* : ∀ *x* t_1 t_2 t_3,

appears_free_in x t_1 →

appears_free_in x ($case$ t_1 t_2 t_3)

| *afi_tcase2* : ∀ *x* t_1 t_2 t_3,

appears_free_in x t_2 →

appears_free_in x ($case$ t_1 t_2 t_3)

| *afi_tcase3* : ∀ *x* t_1 t_2 t_3,

appears_free_in x t_3 →

appears_free_in x ($case$ t_1 t_2 t_3).

`Definition` *closed* (*t*:*term*) := ∀ *x*, ¬ *appears_free_in x t*.

`Lemma` *context_invariance* : ∀ Γ *Gamma' t S n*,

has_type Γ *t S n* →

(∀ *x*, *appears_free_in x t* → Γ *x* = *Gamma' x*) →

has_type Gamma' t S n.

`Lemma` *free_in_context* : ∀ *x t T* Γ *n*,

appears_free_in x t →

has_type Γ *t T n* →

(∃ *T'*, Γ *x* = *Some T'*).

A.4 Reduction

A.4.1 Lambda-calculus reduction relation

$$\texttt{Inductive}\ \mathit{step} : \mathit{term} \to \mathit{term} \to \texttt{Prop} :=$$

$$\mid \mathit{ST\_AppAbs} : \forall\ x\ T\ t\ v,$$
$$\mathit{value}\ v \to$$
$$(\mathit{tapp}\ (\lambda\ x\ T\ t)\ v) \rightsquigarrow [x{:=}v]t$$

$$\mid \mathit{ST\_App1} : \forall\ t_1\ t_1'\ t_2,$$
$$t_1 \rightsquigarrow t_1' \to$$
$$\mathit{tapp}\ t_1\ t_2 \rightsquigarrow \mathit{tapp}\ t_1'\ t_2$$

$$\mid \mathit{ST\_App2} : \forall\ v\ t_2\ t_2',$$
$$\mathit{value}\ v \to$$
$$t_2 \rightsquigarrow t_2' \to$$
$$\mathit{tapp}\ v\ t_2 \rightsquigarrow \mathit{tapp}\ v\ t_2'$$

$$\mid \mathit{ST\_Pair1} : \forall\ t_1\ t_1'\ t_2,$$
$$t_1 \rightsquigarrow t_1' \to$$
$$\mathit{tpair}\ t_1\ t_2 \rightsquigarrow \mathit{tpair}\ t_1'\ t_2$$

$$\mid \mathit{ST\_Pair2} : \forall\ v\ t_2\ t_2',$$
$$\mathit{value}\ v \to$$
$$t_2 \rightsquigarrow t_2' \to$$
$$\mathit{tpair}\ v\ t_2 \rightsquigarrow \mathit{tpair}\ v\ t_2'$$

$$\mid \mathit{ST\_Pi1} : \forall\ v_1\ v_2,$$
$$\mathit{value}\ v_1 \to$$
$$\mathit{value}\ v_2 \to$$
$$\pi_1\ (\mathit{tpair}\ v_1\ v_2) \rightsquigarrow v_1$$

$$\mid \mathit{ST\_Pi2} : \forall\ v_1\ v_2,$$
$$\mathit{value}\ v_1 \to$$

$$value\ v_2 \to$$

$$\pi_2\ (tpair\ v_1\ v_2) \rightsquigarrow v_2$$

| *ST_Pi1E* : $\forall\ t\ t'$,

$$t \rightsquigarrow t' \to$$

$$\pi_1\ t \rightsquigarrow \pi_1\ t'$$

| *ST_Pi2E* : $\forall\ t\ t'$,

$$t \rightsquigarrow t' \to$$

$$\pi_2\ t \rightsquigarrow \pi_2\ t'$$

| *ST_InL* : $\forall\ t_1\ T\ t_1'$,

$$t_1 \rightsquigarrow t_1' \to$$

$$inl\ t_1\ T \rightsquigarrow inl\ t_1'\ T$$

| *ST_InR* : $\forall\ t_1\ T\ t_1'$,

$$t_1 \rightsquigarrow t_1' \to$$

$$inr\ t_1\ T \rightsquigarrow inr\ t_1'\ T$$

| *ST_Case* : $\forall\ t_1\ t_1'\ t_2\ t_3$,

$$t_1 \rightsquigarrow t_1' \to$$

$$case\ t_1\ t_2\ t_3 \rightsquigarrow case\ t_1'\ t_2\ t_3$$

| *ST_CaseL* : $\forall\ v_1\ T\ t_2\ t_3$,

$$value\ v_1 \to$$

$$case\ (inl\ v_1\ T)\ t_2\ t_3 \rightsquigarrow tapp\ t_2\ v_1$$

| *ST_CaseR* : $\forall\ v_1\ T\ t_2\ t_3$,

$$value\ v_1 \to$$

$$case\ (inr\ v_1\ T)\ t_2\ t_3 \rightsquigarrow tapp\ t_3\ v_1$$

`where` "t1 '~>' t2" := (*step* $t_1\ t_2$).

`Inductive` *nstep* : *term* → *term* → *nat* → `Prop` :=

| *nstep_refl* : $\forall$ (*t* : *term*),

$$t \rightsquigarrow^\star t\ //\ O$$

| *nstep_step* : ∀ (*n* : *nat*) (*t u v*: *term*),

$t \rightsquigarrow u \rightarrow$

$u \rightsquigarrow^\star v \,//\, n \rightarrow$

$t \rightsquigarrow^\star v \,//\, (S\ n)$

where "t1 '~»' t2 '//' n" := (*nstep* $t_1\ t_2\ n$).

Theorem *type_unique* : ∀ *t* Γ $T_1\ T_2\ n\ m$,

has_type Γ *t* $T_1\ n \rightarrow$

has_type Γ *t* $T_2\ m \rightarrow (T_1 = T_2 \wedge n = m)$.

Theorem *typable_empty_everywhere*:

∀ Γ *T t n*,

has_type empty t T n → *has_type* Γ *t T n*.

Lemma *values_irreducible* : ∀ *v*, *value v* → ∀ *t*, ~(*v* ⇝ *t*).

Theorem *complexity_of_well_typed_values* :

∀ *t T n*,

has_type empty t T n → *value t* → *n* = *O*.

Lemma *values_subst_preserves_typing* : ∀ Γ *x U v t T n m*,

has_type (*extend* Γ *x U*) *t T n* →

has_type empty v U m →

value v →

has_type Γ ([*x*:=*v*] *t*) *T n*.

Corollary *typable_empty_closed* : ∀ *t T n*,

has_type empty t T n →

closed t.

Lemma *canonical_forms_pair* : ∀ *t* $T_1\ T_2\ n$,

has_type empty t (*TProd* $T_1\ T_2$) *n* →

value t →

∃ $t_1\ t_2$, *t* = *tpair* $t_1\ t_2$.

Lemma *canonical_forms_sum* : $\forall$ *t* T_1 T_2 *n*,

has_type empty t (*TSum* T_1 T_2) *n* $\rightarrow$

value t $\rightarrow$

($\exists$ *t'*, *t* = inl *t'* T_2) $\vee$ ($\exists$ *t'*, *t* = inr *t'* T_1).

Lemma *canonical_forms_tunit* : $\forall$ *t n*,

has_type empty t () *n* $\rightarrow$

value t $\rightarrow$

t = nil $\wedge$ *n* = *O*.

Progress

Theorem progress : $\forall$ *t T n*,

has_type empty t T n $\rightarrow$ *value t* $\vee$ $\exists$ *t'*, *t* $\rightsquigarrow$ *t'*.

Corollary *not_value_step* : $\forall$ *t T n*,

has_type empty t T n $\rightarrow$ $\neg$ *value t* $\rightarrow$ $\exists$ *t'*, *t* $\rightsquigarrow$ *t'*.

Preservation

Theorem *preservation* :

$\forall$ *t t' T n*,

has_type empty t T n $\rightarrow$

t $\rightsquigarrow$ *t'* $\rightarrow$

$\exists$ *m*, *m* < *n* $\wedge$ *has_type empty t' T m*.

Corollary *preservation_nstep* :

$\forall$ *n m t t' T*,

has_type empty t T n $\rightarrow$

t $\rightsquigarrow^{\star}$ *t'* // *m* $\rightarrow$

$\exists$ *o*, *o* $\leq$ (*n* $-$ *m*) $\wedge$

has_type empty t' T o.

Determinism

Theorem *step_deterministic* : $\forall\ t\ u\ v$,

$$t \rightsquigarrow u \rightarrow$$

$$t \rightsquigarrow v \rightarrow$$

$$u = v.$$

A.4.2 Congruence Lemmmas

Lemma *Congruence_AppAbs* : $\forall\ (x : id)\ (T : Ty)\ (t\ v : term)$,

value $v \rightarrow$

$(\backslash(x{:}T)\ t)\ \$\ v \rightsquigarrow^{\star}\ [x := v]\ t\ //\ 1.$

Lemma *Congruence_App1* : $\forall\ t_1\ t_1'\ t_2\ n$,

$(t_1 \rightsquigarrow^{\star}\ t_1'\ //\ n) \rightarrow$

$(tapp\ t_1\ t_2) \rightsquigarrow^{\star}\ (tapp\ t_1'\ t_2)\ //\ n.$

Lemma *Congruence_App2* : $\forall\ v\ t\ t'\ n$,

value $v \rightarrow$

$(t \rightsquigarrow^{\star}\ t'\ //\ n) \rightarrow$

$(tapp\ v\ t) \rightsquigarrow^{\star}\ (tapp\ v\ t')\ //\ n.$

Lemma *Congruence_Pair1* : $\forall\ t_1\ t_1'\ t_2\ n$,

$(t_1 \rightsquigarrow^{\star}\ t_1'\ //\ n) \rightarrow$

$(tpair\ t_1\ t_2) \rightsquigarrow^{\star}\ (tpair\ t_1'\ t_2)\ //\ n.$

Lemma *Congruence_Pair2* : $\forall\ v_1\ t_2\ t_2'\ n$,

value $v_1 \rightarrow$

$t_2 \rightsquigarrow^{\star}\ t_2'\ //\ n \rightarrow$

$(tpair\ v_1\ t_2) \rightsquigarrow^{\star}\ (tpair\ v_1\ t_2')\ //\ n.$

Lemma *Congruence_Pi1* : $\forall\ v_1\ v_2$,

value $v_1 \rightarrow$

value $v_2 \rightarrow$

$\pi_1\ (tpair\ v_1\ v_2) \rightsquigarrow^\star v_1\ //\ 1.$

Lemma *Congruence_Pi2* : $\forall\ v_1\ v_2$,

value $v_1 \rightarrow$

value $v_2 \rightarrow$

$\pi_2\ (tpair\ v_1\ v_2) \rightsquigarrow^\star v_2\ //\ 1.$

Lemma *Congruence_Pi1E*: $\forall\ t\ t'\ n$,

$t \rightsquigarrow^\star t'\ //\ n \rightarrow$

$(\pi_1\ t \rightsquigarrow^\star \pi_1\ t'\ //\ n)$.

Lemma *Congruence_Pi2E*: $\forall\ t\ t'\ n$,

$t \rightsquigarrow^\star t'\ //\ n \rightarrow$

$(\pi_2\ t \rightsquigarrow^\star \pi_2\ t'\ //\ n)$.

Lemma *Congruence_Tinl* : $\forall\ T\ t_1\ t_2\ n$,

$t_1 \rightsquigarrow^\star t_2\ //\ n \rightarrow (inl\ t_1\ T) \rightsquigarrow^\star (inl\ t_2\ T)\ //\ n.$

Lemma *Congruence_Tinr* : $\forall\ T\ t_1\ t_2\ n$,

$t_1 \rightsquigarrow^\star t_2\ //\ n \rightarrow (inr\ t_1\ T) \rightsquigarrow^\star (inr\ t_2\ T)\ //\ n.$

Lemma *Congruence_Tcase* : $\forall\ t_1\ t_1'\ t_2\ t_3\ n$,

$t_1 \rightsquigarrow^\star t_1'\ //\ n \rightarrow case\ t_1\ t_2\ t_3 \rightsquigarrow^\star case\ t_1'\ t_2\ t_3\ //\ n.$

Lemma *Congruence_ST_CaseL* : $\forall\ T\ v_1\ t_2\ t_3$,

value $v_1 \rightarrow$

$case\ (inl\ v_1\ T)\ t_2\ t_3 \rightsquigarrow^\star tapp\ t_2\ v_1\ //\ 1.$

Lemma *Congruence_ST_CaseR* : $\forall\ T\ v_1\ t_2\ t_3$,

value $v_1 \rightarrow$

$case\ (inr\ v_1\ T)\ t_2\ t_3 \rightsquigarrow^\star tapp\ t_3\ v_1\ //\ 1.$

Theorem *id_not_step* : $\forall\ i\ (u : term)$,

$\neg\ (var\ i \rightsquigarrow u)$.

Corollary *id_not_nstep'* : $\forall\ i\ (u : term)$,

$\neg\ (var\ i \rightsquigarrow^\star u\ //\ 1)$.

Corollary *id_not_nstep* : $\forall$ *i* (*u* :*term*) *n*,

$var\ i \neq u \rightarrow$

$\neg\ (var\ i \rightsquigarrow^{\star}\ u\ //\ n)$.

A.4.3 Inversion Principles

Lemma *var_inv1* : $\forall$ (*i* : *id*) *u*,

i $\rightsquigarrow$ *u* $\rightarrow$ *u* = *i*.

Lemma *var_inv2* : $\forall$ (*i* : *id*) *u*,

$\neg$ (*i* $\rightsquigarrow$ *u*).

Lemma *pair_inv1* : $\forall$ *l r l' r'*,

tpair l r $\rightsquigarrow$ *tpair l' r'* $\rightarrow$

$\neg$ *value l* $\rightarrow$

(*l* $\rightsquigarrow$ *l'*).

Lemma *pair_inv2* : $\forall$ *l r l' r'*,

tpair l r $\rightsquigarrow$ *tpair l' r'* $\rightarrow$

value l $\rightarrow$

(*r* $\rightsquigarrow$ *r'*).

Lemma *pair_inv_l* : $\forall$ *l r p*,

tpair l r $\rightsquigarrow$ *p* $\rightarrow$

$\neg$ *value l* $\rightarrow$

$\exists$ *l'*, *l* $\rightsquigarrow$ *l'*.

Lemma *pair_inv_l'* : $\forall$ *l r l' r'*,

tpair l r $\rightsquigarrow$ *tpair l' r'* $\rightarrow$

$\neg$ *value l* $\rightarrow$

l $\rightsquigarrow$ *l'*.

Lemma *pair_inv_r* : $\forall$ *l r p*,

tpair l r $\rightsquigarrow$ *p* $\rightarrow$

$value\ l \to$

$\exists\ r',\ r \rightsquigarrow r'.$

Lemma $pair\_step\_pair : \forall\ l\ r\ p,$

$tpair\ l\ r \rightsquigarrow p \to$

$\exists\ l'\ r',\ p = (tpair\ l'\ r').$

Lemma $app\_inv1 : \forall\ m\ n\ p,$

$m\ \$\ n \rightsquigarrow p \to$

$\neg\ value\ m \to$

$(\exists\ n',\ n \rightsquigarrow n') \vee (\exists\ m',\ m \rightsquigarrow m').$

Lemma $app\_inv2 : \forall\ m\ n\ p,$

$m\ \$\ n \rightsquigarrow p \to$

$\neg\ value\ n \to$

$(\exists\ n',\ n \rightsquigarrow n') \vee (\exists\ m',\ m \rightsquigarrow m').$

Lemma $app\_inv3 : \forall\ m\ n\ p,$

$m\ \$\ n \rightsquigarrow p \to$

$value\ m \to$

$\neg\ value\ n \to$

$(\exists\ n',\ n \rightsquigarrow n').$

Lemma $app\_inv4 : \forall\ m\ n\ p,$

$m\ \$\ n \rightsquigarrow p \to$

$\neg\ value\ m \to$

$\neg\ value\ n \to$

$(\exists\ m',\ m \rightsquigarrow m').$

Lemma $app\_abs\_inv : \forall\ m\ n\ p,$

$m\ \$\ n \rightsquigarrow p \to$

$value\ m \to$

$value\ n \to$

$\neg$ ($\exists$ *m'*, *m* $\rightsquigarrow$ *m'*).

Lemma *val_inv* : $\forall$ *v u n*,

value v $\rightarrow$

v $\rightsquigarrow^\star$ *u* // *n* $\rightarrow$ *v* = *u*.

Lemma *step_value_second* : $\forall$ *t t' v n*,

value v $\rightarrow$

t $\rightsquigarrow$ *t'* $\rightarrow$

t $\rightsquigarrow^\star$ *v* // *n* $\rightarrow$

t' $\rightsquigarrow^\star$ *v* // (*n* $-$ 1).

Lemma *step_value_unique* : $\forall$ *t v v' n m*,

value v $\rightarrow$

value v' $\rightarrow$

t $\rightsquigarrow^\star$ *v* // *n* $\rightarrow$

t $\rightsquigarrow^\star$ *v'* // *m* $\rightarrow$

v = *v'*.

Lemma *step_same_value* : $\forall$ *t t' v n m*,

value v $\rightarrow$

t $\rightsquigarrow^\star$ *t'* // *n* $\rightarrow$

t $\rightsquigarrow^\star$ *v* // *m* $\rightarrow$

t' $\rightsquigarrow^\star$ *v* // (*m* $-$ *n*).

Lemma *step_values_same* : $\forall$ *t v n m*,

value v $\rightarrow$

t $\rightsquigarrow^\star$ *v* // *n* $\rightarrow$

t $\rightsquigarrow^\star$ *v* // *m* $\rightarrow$

(*m* = *n*).

Lemma *step_values_same'* : $\forall$ *t v v' n m*,

value v $\rightarrow$

$$value\ v' \to$$

$$t \rightsquigarrow^\star v \,//\, n \to$$

$$t \rightsquigarrow^\star v' \,//\, m \to$$

$$(m = n).$$

.

```
Definition normal_form (t : term) : Prop :=
  ¬ ∃ t', t ⇝ t'.

Lemma value_normal_form : ∀ (v : term),
                            value v → normal_form v.

Inductive terminates : term → Prop :=
| terminates_intro : ∀ t n v, t ⇝* v // n → value v → terminates t.

Definition terminates' (t : term) :=
  ∃ v n, t ⇝* v // n ∧ value v.

Theorem teqt : ∀ t, terminates t ↔ terminates' t.

Lemma values_terminate : ∀ v,
      value v → terminates v.
```

A.5 Reducibility

```
Fixpoint R (T:Ty) (t:term) {struct T} : Prop :=
  (∃ n, has_type empty t T n) ∧ (terminates t) ∧
  (match T with
   | TArrow m T1 T2 ⇒ ∀ s,
                        R T1 s → R T2 (tapp t s)
   | TProd T1 T2 ⇒ ∃ t1 t2 o,
```

$t \rightsquigarrow^\star (tpair\ t_1\ t_2)\ //\ o \wedge$

$value\ t_1 \wedge$

$value\ t_2 \wedge$

$R\ T_1\ t_1 \wedge$

$R\ T_2\ t_2$

| $TSum\ T_1\ T_2 \Rightarrow \exists\ t'\ o,$

$value\ t' \wedge$

$((t \rightsquigarrow^\star inl\ t'\ T_2\ //\ o \wedge R\ T_1\ t') \vee$

$(t \rightsquigarrow^\star inr\ t'\ T_1\ //\ o \wedge R\ T_2\ t'))$

| $TNil \Rightarrow True$

end).

Theorem $R\_terminates : \forall\ T\ t,$

$R\ T\ t \rightarrow terminates\ t.$

Theorem $R\_typable\_empty : \forall\ T\ t,$

$R\ T\ t \rightarrow \exists\ n,\ has\_type\ empty\ t\ T\ n.$

Lemma $step\_preserves\_termination : \forall\ t\ t',$

$(t \rightsquigarrow t') \rightarrow (terminates\ t \leftrightarrow terminates\ t').$

Lemma $nstep\_preserves\_termination : \forall\ t\ t'\ n,$

$(t \rightsquigarrow^\star t'\ //\ n) \rightarrow (terminates\ t \leftrightarrow terminates\ t').$

Lemma $step\_preserves\_R : \forall\ T\ t\ t',$

$t \rightsquigarrow t' \rightarrow$

$R\ T\ t \rightarrow$

$R\ T\ t'.$

Lemma $nstep\_preserves\_R : \forall\ T\ t\ t'\ n,$

$t \rightsquigarrow^\star t'\ //\ n \rightarrow$

$R\ T\ t \rightarrow$

```
                                        R T t'.

Lemma step_preserves_R' : ∀ T t t' n,
        has_type empty t T n →
                        t ⇝ t' →
                        R T t' →
                        R T t.

Lemma nstep_preserves_R' : ∀ T t t' n m,
        has_type empty t T n →
                        t ⇝* t' // m →
                        R T t' →
                        R T t.

Definition env := list (id × term).

Fixpoint closed_env (env:env) {struct env} :=
          match env with
          | nil ⇒ True
          | (x,t)::env' ⇒ closed t ∧ closed_env env'
          end.

Fixpoint msubst (ss:env) (t:term) {struct ss} : term :=
          match ss with
          | nil ⇒ t
          | ((x,s)::ss') ⇒ msubst ss' ([x:=s]t)
          end.

Definition tass := list (id × Ty).

Fixpoint mextend (Γ : context) (xts : tass) :=
          match xts with
          | nil ⇒ Γ
```

```
    | ((x,v)::xts') ⇒ extend (mextend Γ xts') x v
    end.

Fixpoint lookup {X:Set} (k : id) (l : list (id × X)) {struct l} : option X :=
    match l with
    | nil ⇒ None
    | (j,x) :: l' ⇒ if eq_id_dec j k then Some x else lookup k l'
    end.

Fixpoint drop {X:Set} (n:id) (nxs:list (id × X)) {struct nxs} : list (id × X) :=
    match nxs with
    | nil ⇒ nil
    | ((n',x)::nxs') ⇒ if eq_id_dec n' n then drop n nxs' else (n',x)::(drop n nxs')
    end.

Inductive instantiation : tass → env → Prop :=
  | V_nil : instantiation nil nil
  | V_cons : ∀ x T v c e,
                          value v →
                            R T v →
          instantiation c e →
          instantiation ((x,T)::c) ((x,v)::e).

Lemma mextend_lookup : ∀ (c:tass) (x:id),
    lookup x c = (mextend empty c) x.

Lemma mextend_drop : ∀ (c:tass) Γ x x',
    mextend Γ (drop x c) x' = if eq_id_dec x x' then Γ x' else mextend Γ c x'.
```

Properties of Instantiations

```
Lemma instantiation_domains_match: ∀ {c} {e},
```

instantiation c e →

∀ {x} {T}, lookup x c = Some T → ∃ t, lookup x e = Some t.

Lemma *instantiation_env_closed : ∀ c e,*

instantiation c e → closed_env e.

Lemma *instantiation_R : ∀ c e,*

instantiation c e →

∀ x t T,

lookup x c = Some T →

lookup x e = Some t →

R T t.

Lemma *instantiation_drop : ∀ c env,*

instantiation c env → ∀ x, instantiation (drop x c) (drop x env).

Lemma *mextend_empty_lookup : ∀ c x, (mextend empty c) x = lookup x c.*

Lemma *msubst_closed: ∀ t,*

closed t →

∀ ss,

msubst ss t = t.

Lemma *msubst_preserves_typing : ∀ c e,*

instantiation c e →

∀ Γ t S n, has_type (mextend Γ c) t S n →

has_type Γ (msubst e t) S n.

Lemma *subst_msubst: ∀ env x v t,*

closed v →

closed_env env →

msubst env ([x:=v]t) = [x:=v](msubst (drop x env) t).

Lemma *msubst_var: ∀ ss x, closed_env ss →*

msubst ss (var *x*) =

match *lookup x ss* with

| *Some t* ⇒ *t*

| *None* ⇒ var *x*

end.

Lemma *msubst_abs*: ∀ *ss x T t*,

msubst ss (λ *x T t*) = λ *x T* (*msubst* (*drop x ss*) *t*).

Lemma *msubst_app* : ∀ *ss* t_1 t_2,

msubst ss (*tapp* t_1 t_2) = *tapp* (*msubst ss* t_1) (*msubst ss* t_2).

Lemma *msubst_pair* : ∀ *ss* t_1 t_2,

msubst ss (*tpair* t_1 t_2) = *tpair* (*msubst ss* t_1) (*msubst ss* t_2).

Lemma *msubst_pi1* : ∀ *ss* t_1,

msubst ss (π_1 t_1) = π_1 (*msubst ss* t_1).

Lemma *msubst_pi2* : ∀ *ss* t_1,

msubst ss (π_2 t_1) = π_2 (*msubst ss* t_1).

Lemma *msubst_tinl* : ∀ *ss T t*,

msubst ss (inl *t T*) = inl (*msubst ss t*) *T*.

Lemma *msubst_tinr* : ∀ *ss T t*,

msubst ss (inr *t T*) = inr (*msubst ss t*) *T*.

Lemma *msubst_tcase* : ∀ *ss* t_1 t_2 t_3,

msubst ss ($case$ t_1 t_2 t_3) = $case$ (*msubst ss* t_1) (*msubst ss* t_2) (*msubst ss* t_3).

Lemma *msubst_tunit* : ∀ *ss*,

msubst ss nil = nil.

Lemma *msubst_R* : ∀ *c env t T n*,

has_type (*mextend empty c*) *t T n* →

instantiation c env →

R T (*msubst env t*).

Theorem *normalization* : ∀ (*T*:*Ty*) (*t*:*term*) (*n*:*nat*),

has_type empty t T n → *terminates t.*

Theorem *normalization'* : ∀ (*T*:*Ty*) (*t*:*term*) (*n*:*nat*),

has_type empty t T n → *terminates' t.*

Appendix B

RWC COQ Code

B.1 Syntax

Here is the syntax for ReWire.

B.1.1 Monads and Types

Here is the syntax for layered state monads.

```
Inductive Eff : Type :=
  | EffNone : Eff
  | EffR : Eff
  | EffW : Eff
  | EffRW : Eff.

Inductive SMo : Type :=
  | MIdentity : SMo
  | MStateT : Ty → Eff → SMo → SMo


with Mo : Type :=
  | MReactT : Ty → Ty → SMo → Mo
  | MNonReact : SMo → Mo
```

Here is the syntax for "ordinary" types.

```
with Ty : Type :=
  | TArrow : Ty → Ty → Ty
  | TProd : Ty → Ty → Ty
  | TSum : Ty → Ty → Ty
  | TNil : Ty
  | TMonadic : Mo → Ty → Ty.
```

Equality of Eff, Ty, Mo, SMo is deciable:

```
Lemma Eff_eq_dec : ∀ e e' : Eff, {e = e'} + {e ≠ e'}.

Lemma Ty_eq_dec : ∀ x y : Ty, {x = y} + {x ≠ y}
 with Mo_eq_dec : ∀ m n : Mo, {m = n} + {m ≠ n}
 with SMo_eq_dec : ∀ s r : SMo, {s = r} + {s ≠ r}.
```

Mutual induction schemes for types and monads will be useful.

```
Scheme Ty_Mo_SMo_ind := Induction for Ty Sort Prop
  with Mo_Ty_SMo_ind := Induction for Mo Sort Prop
  with SMo_Ty_Mo_ind := Induction for SMo Sort Prop.

Definition on_io (P:Ty → Prop) (M:Mo) : Prop :=
  match M with
  | MNonReact SM ⇒ True
  | MReactT TI TO SM ⇒ P TI ∧ P TO
  end.

Theorem Ty_Mo_SMo_mutind_better'
      : ∀ (P : Ty → Prop) (P0 : Mo → Prop) (P1 : SMo → Prop),
        (∀ T1 : Ty, P T1 → ∀ T2 : Ty, P T2 → P (TArrow T1 T2)) →
        (∀ T1 : Ty, P T1 → ∀ T2 : Ty, P T2 → P (TProd T1 T2)) →
        (∀ T1 : Ty, P T1 → ∀ T2 : Ty, P T2 → P (TSum T1 T2)) →
        P TNil →
```

$(\forall\ M : Mo, (P0\ M \wedge on\_io\ P\ M) \rightarrow$

$\forall\ T : Ty, P\ T \rightarrow$

$P\ (TMonadic\ M\ T)) \rightarrow$

$(\forall\ TI : Ty,$

$P\ TI \rightarrow$

$\forall\ TO : Ty, P\ TO \rightarrow \forall\ SM : SMo, P1\ SM \rightarrow P0\ (MReactT\ TI\ TO\ SM)) \rightarrow$

$(\forall\ SM : SMo, P1\ SM \rightarrow P0\ (MNonReact\ SM)) \rightarrow$

$P1\ MIdentity \rightarrow$

$(\forall\ T : Ty,$

$P\ T \rightarrow \forall\ (E : Eff)\ (SM : SMo), P1\ SM \rightarrow P1\ (MStateT\ T\ E\ SM)) \rightarrow$

$(\forall\ T : Ty, P\ T) \wedge (\forall\ M : Mo, P0\ M \wedge on\_io\ P\ M) \wedge (\forall\ SM : SMo, P1\ SM).$

`Theorem` $Ty\_Mo\_SMo\_mutind\_better$

$: \forall\ (P : Ty \rightarrow \texttt{Prop})\ (P0 : Mo \rightarrow \texttt{Prop})\ (P1 : SMo \rightarrow \texttt{Prop}),$

$(\forall\ T1 : Ty, P\ T1 \rightarrow \forall\ T2 : Ty, P\ T2 \rightarrow P\ (TArrow\ T1\ T2)) \rightarrow$

$(\forall\ T1 : Ty, P\ T1 \rightarrow \forall\ T2 : Ty, P\ T2 \rightarrow P\ (TProd\ T1\ T2)) \rightarrow$

$(\forall\ T1 : Ty, P\ T1 \rightarrow \forall\ T2 : Ty, P\ T2 \rightarrow P\ (TSum\ T1\ T2)) \rightarrow$

$P\ TNil \rightarrow$

$(\forall\ M : Mo, (P0\ M \wedge on\_io\ P\ M) \rightarrow$

$\forall\ T : Ty, P\ T \rightarrow$

$P\ (TMonadic\ M\ T)) \rightarrow$

$(\forall\ TI : Ty,$

$P\ TI \rightarrow$

$\forall\ TO : Ty, P\ TO \rightarrow \forall\ SM : SMo, P1\ SM \rightarrow P0\ (MReactT\ TI\ TO\ SM)) \rightarrow$

$(\forall\ SM : SMo, P1\ SM \rightarrow P0\ (MNonReact\ SM)) \rightarrow$

$P1\ MIdentity \rightarrow$

$(\forall\ T : Ty,$

$P\ T \rightarrow \forall\ (E : Eff)\ (SM : SMo), P1\ SM \rightarrow P1\ (MStateT\ T\ E\ SM)) \rightarrow$

$$(\forall\ T : Ty,\ P\ T) \wedge (\forall\ M : Mo,\ P0\ M) \wedge (\forall\ SM : SMo,\ P1\ SM).$$

Relating state monads by permissiveness

```
Inductive Eff_lt : Eff → Eff → Prop :=
  | Eff_lt_None_None : Eff_lt EffNone EffNone
  | Eff_lt_None_R : Eff_lt EffNone EffR
  | Eff_lt_None_W : Eff_lt EffNone EffW
  | Eff_lt_None_RW : Eff_lt EffNone EffRW
  | Eff_lt_R_R : Eff_lt EffR EffR
  | Eff_lt_R_RW : Eff_lt EffR EffRW
  | Eff_lt_W_W : Eff_lt EffW EffW
  | Eff_lt_W_RW : Eff_lt EffW EffRW
  | Eff_lt_RW_RW : Eff_lt EffRW EffRW.

Theorem Eff_lt_refl : ∀ E, Eff_lt E E.

Theorem Eff_lt_antisymm : ∀ E1 E2,
                          Eff_lt E1 E2 →
                          Eff_lt E2 E1 →
                          E1 = E2.

Theorem Eff_lt_trans : ∀ E1 E2 E3,
                       Eff_lt E1 E2 →
                       Eff_lt E2 E3 →
                       Eff_lt E1 E3.
```

Partial order on state monads. M1 less permissive than M2 means that M1 has an identical store shape but fewer or the same permissions.

```
Inductive smo_less_permissive : SMo → SMo → Prop :=
```

```
  | LP_StateT : ∀ T E1 E2 M1 M2,
                Eff_lt E1 E2 →
                smo_less_permissive M1 M2 →
                smo_less_permissive (MStateT T E1 M1)
                                    (MStateT T E2 M2)
  | LP_Identity : smo_less_permissive MIdentity MIdentity.

Theorem less_permissive_refl : ∀ M, smo_less_permissive M M.

Theorem less_permissive_antisymm : ∀ M1 M2, smo_less_permissive M1 M2 →
                                            smo_less_permissive M2 M1
→
                                            M1 = M2.

Theorem less_permissive_trans: ∀ M1 M2 M3, smo_less_permissive M1 M2 →
                                           smo_less_permissive M2 M3
→
                                           smo_less_permissive M1 M3.

Inductive Eff_disjoint : Eff → Eff → Prop :=
  | Eff_disjoint_None_None : Eff_disjoint EffNone EffNone
  | Eff_disjoint_None_R : Eff_disjoint EffNone EffR
  | Eff_disjoint_None_W : Eff_disjoint EffNone EffW
  | Eff_disjoint_None_RW : Eff_disjoint EffNone EffRW
  | Eff_disjoint_R_None : Eff_disjoint EffR EffNone
  | Eff_disjoint_R_W : Eff_disjoint EffR EffW
  | Eff_disjoint_W_None : Eff_disjoint EffW EffNone
  | Eff_disjoint_W_R : Eff_disjoint EffW EffR
  | Eff_disjoint_RW_None : Eff_disjoint EffRW EffNone.

Theorem Eff_disjoint_symm : ∀ E1 E2,
```

Eff_disjoint E1 E2 →

Eff_disjoint E2 E1.

`Inductive` *smo_disjoint* : *SMo → SMo →* `Prop` :=

| *Disjoint_StateT* : ∀ *T E1 E2 M1 M2*,

smo_disjoint M1 M2 →

Eff_disjoint E1 E2 →

smo_disjoint (*MStateT T E1 M1*)

(*MStateT T E2 M2*)

| *Disjoint_Identity* : *smo_disjoint MIdentity MIdentity.*

`Theorem` *smo_disjoint_symm* : ∀ *M1 M2*, *smo_disjoint M1 M2 →*

smo_disjoint M2 M1.

B.1.2 Terms and Configurations

`Inductive` *tm* : `Type` :=

| *tvar* : *id → tm*

| *tapp* : *tm → tm → tm*

| *tabs* : *id → Ty → tm → tm*

| *tunit* : *tm*

| *tpair* : *tm → tm → tm*

| *tproj* : *tm → tm → tm*

| *tinl* : *Ty → tm → tm*

| *tinr* : *Ty → tm → tm*

| *tcase* : *tm → tm → tm → tm*

| *treturn* : *Mo → tm → tm*

| *tbind* : *Ty* → *tm* → *tm* → *tm*

| *tlift* : *Mo* → *tm* → *tm*

| *televate* : *SMo* → *tm* → *tm*

| *tget* : *SMo* → *tm*

| *tput* : *SMo* → *tm* → *tm*

| *trunst* : *tm* → *tm* → *tm*

| *trunid* : *tm* → *tm*

| *tpause* : *Mo* → *Ty* → *tm* → *tm*

| *tunfold* : *Mo* → *Ty* → *Ty* → *tm* → *tm* → *tm*

| *trunre* : *Ty* → *tm* → *tm*.

The next lemma aims to address the naming conventions and subcases of the destruct tactic, to wit, they suck.

```
Infix "$" := tapp (at level 40).

Notation "f »=[ T ] g" := (tbind T f g)
                                    (at level 40, T at level 99, format "'[hv ' f »=[ T
] g ']'").

Notation "\( x ':' T ) t" := (tabs x T t)
                                    (at level 40, x at level 99, format "'[hv ' \( x ':' T
) t ']'").

Lemma term_cases : ∀ P : tm → Prop,
      (∀ i : id, P (tvar i)) →
      (∀ t1 t2 : tm, P (tapp t1 t2)) →
      (∀ (i : id) (T : Ty) (t : tm), P (tabs i T t)) →
      P tunit →
```

$$(\forall\ t1\ t2 : tm,$$

$$P\ (tpair\ t1\ t2)) \rightarrow$$

$$(\forall\ t1\ t2 : tm,$$

$$P\ (tproj\ t1\ t2)) \rightarrow$$

$$(\forall\ (T : Ty)\ (t : tm),$$

$$P\ (tinl\ T\ t)) \rightarrow$$

$$(\forall\ (T : Ty)\ (t : tm),$$

$$P\ (tinr\ T\ t)) \rightarrow$$

$$(\forall\ t1\ t2\ t3 : tm,$$

$$P\ (tcase\ t1\ t2\ t3)) \rightarrow$$

$$(\forall\ (M : Mo)\ (t : tm),$$

$$P\ (treturn\ M\ t)) \rightarrow$$

$$(\forall\ (T : Ty)\ (t1\ t2 : tm),$$

$$P\ (tbind\ T\ t1\ t2)) \rightarrow$$

$$(\forall\ (M : Mo)\ (t : tm),$$

$$P\ (tlift\ M\ t)) \rightarrow$$

$$(\forall\ (S : SMo)\ (t : tm),$$

$$P\ (televate\ S\ t)) \rightarrow$$

$$(\forall\ S : SMo,$$

$$P\ (tget\ S)) \rightarrow$$

$$(\forall\ (S : SMo)\ (t : tm),$$

$$P\ (tput\ S\ t)) \rightarrow$$

$$(\forall\ t1\ t2 : tm,$$

$$P\ (trunst\ t1\ t2)) \rightarrow$$

$$(\forall\ t : tm,$$

$$P\ (trunid\ t)) \rightarrow$$

$$(\forall\ (M : Mo)\ (T : Ty)\ (t : tm),$$

```
                                P (tpause M T t)) →
      (∀ (M : Mo) (TA TB : Ty) (t1 t2 : tm),
                                P (tunfold M TA TB t1 t2)) →
      (∀ (T : Ty) (t : tm),
                                P (trunre T t)) →
      ∀ (t : tm), P t.

Tactic Notation "term_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "tvar" | Case_aux c "tapp" | Case_aux c "tabs"
  | Case_aux c "tunit" | Case_aux c "tpair" | Case_aux c "tproj"
  | Case_aux c "tinl" | Case_aux c "tinr" | Case_aux c "tcase"
  | Case_aux c "treturn" | Case_aux c "tbind" | Case_aux c "tlift"
  | Case_aux c "televate" | Case_aux c "tget" | Case_aux c "tput"
  | Case_aux c "trunst" | Case_aux c "trunid" | Case_aux c "tpause"
  | Case_aux c "tunfold" | Case_aux c "trunre" ].

Definition store := list tm.

Definition configuration := (tm×store)%type.
```

B.2 Lambda Calculus Values

```
Inductive value : tm → Prop :=
  | v_abs : ∀ x T t,
      value (tabs x T t)
  | v_unit : value tunit
  | v_pair : ∀ t1 t2,
      value t1 →
      value t2 →
```

$value\ (tpair\ t1\ t2)$

$|\ v\_inl : \forall\ T\ t,$

$value\ t \rightarrow$

$value\ (tinl\ T\ t)$

$|\ v\_inr : \forall\ T\ t,$

$value\ t \rightarrow$

$value\ (tinr\ T\ t)$

$|\ v\_return : \forall\ M\ t,$

$value\ t \rightarrow$

$value\ (treturn\ M\ t)$

$|\ v\_bind : \forall\ T\ t1\ t2,$

$value\ t1 \rightarrow$

$value\ t2 \rightarrow$

$value\ (tbind\ T\ t1\ t2)$

$|\ v\_lift : \forall\ M\ t,$

$value\ t \rightarrow$

$value\ (tlift\ M\ t)$

$|\ v\_elevate : \forall\ SM\ t,$

$value\ t \rightarrow$

$value\ (televate\ SM\ t)$

$|\ v\_get : \forall\ SM,$

$value\ (tget\ SM)$

$|\ v\_put : \forall\ SM\ t,$

$value\ t \rightarrow$

$value\ (tput\ SM\ t)$

$|\ v\_runst : \forall\ t1\ t2,$

$value\ t1 \rightarrow$

value t2 →

value (*trunst t1 t2*)

| *v_pause* : ∀ *M T t*,

value t →

value (*tpause M T t*)

| *v_unfold* : ∀ *M TA TB t1 t2*,

value t1 →

value t2 →

value (*tunfold M TA TB t1 t2*)

| *v_runre* : ∀ *T t1*,

value t1 →

value (*trunre T t1*).

`Theorem` *value_dec* : ∀ (*t*:*tm*), *value t* ∨ ¬ (*value t*).

B.2.1 Done Configurations

`Inductive` *done_mo* : *configuration* → `Prop` :=

| *done_return* : ∀ *M v Sto*, *value v* → *done_mo* (*treturn M v*,*Sto*)

| *done_pause* : ∀ *M T v Sto*, *value v* → *done_mo* (*tpause M T v*,*Sto*).

`Theorem` *done_mo_dec* (*co*:*configuration*) : *done_mo co* ∨ ¬ (*done_mo co*).

B.3 Typing Judgments

B.3.1 For terms

`Definition context` := *partial_map Ty*.

`Reserved Notation` "Gamma '⊢' t '\in' T" (`at level` 40).

`Inductive` *has_type* : `context` → *tm* → *Ty* → `Prop` :=

| *T_Var* : ∀ Γ *x T*,

Γ *x* = *Some T* → Γ ⊢ *tvar x* : *T*

| *T_Abs* : ∀ Γ *x T T'* *t*,

extend Γ *x T* ⊢ *t* : *T'* →

Γ ⊢ *tabs x T t* : *TArrow T T'*

| *T_App* : ∀ *T T'* Γ *t1 t2*,

Γ ⊢ *t1* : *TArrow T T'* →

Γ ⊢ *t2* : *T* →

Γ ⊢ *tapp t1 t2* : *T'*

| *T_Unit* : ∀ *Gamma*,

Γ ⊢ *tunit* : *TNil*

| *T_Pair* : ∀ Γ *T1 T2 t1 t2*,

Γ ⊢ *t1* : *T1* →

Γ ⊢ *t2* : *T2* →

Γ ⊢ *tpair t1 t2* : *TProd T1 T2*

| *T_Proj* : ∀ Γ *T1 T2 T3 t t'*,

Γ ⊢ *t* : (*TProd T1 T2*) →

Γ ⊢ *t'* : *TArrow T1* (*TArrow T2 T3*) →

Γ ⊢ *tproj t t'* : *T3*

| *T_Inl* : ∀ Γ *T1 T2 t*,

Γ ⊢ *t* : *T1* →

Γ ⊢ *tinl T2 t* : *TSum T1 T2*

| *T_Inr* : ∀ Γ *T1 T2 t*,

Γ ⊢ *t* : *T2* →

Γ ⊢ *tinr T1 t* : *TSum T1 T2*

| *T_Case* : ∀ Γ *Tl Tr Tres ts tl tr*,

Γ ⊢ *ts* : *TSum Tl Tr* →

$\Gamma \vdash tl : TArrow\ Tl\ Tres \rightarrow$

$\Gamma \vdash tr : TArrow\ Tr\ Tres \rightarrow$

$\Gamma \vdash tcase\ ts\ tl\ tr : Tres$

| *T_Return* : $\forall\ \Gamma\ T\ M\ t$,

$\Gamma \vdash t : T \rightarrow$

$\Gamma \vdash treturn\ M\ t : TMonadic\ M\ T$

| *T_Bind* : $\forall\ \Gamma\ t\ M\ T1\ t'\ T2$,

$\Gamma \vdash t : TMonadic\ M\ T1 \rightarrow$

$\Gamma \vdash t' : TArrow\ T1\ (TMonadic\ M\ T2) \rightarrow$

$\Gamma \vdash tbind\ T2\ t\ t' : TMonadic\ M\ T2$

| *T_LiftSt* : $\forall\ \Gamma\ t\ SM\ E\ T\ T'$,

$\Gamma \vdash t : TMonadic\ (MNonReact\ SM)\ T' \rightarrow$

$\Gamma \vdash tlift\ (MNonReact\ (MStateT\ T\ E\ SM))\ t : TMonadic\ (MNonReact\ (MStateT\ T\ E\ SM))\ T'$

| *T_LiftRe* : $\forall\ \Gamma\ t\ SM\ T\ TI\ TO$,

$\Gamma \vdash t : TMonadic\ (MNonReact\ SM)\ T \rightarrow$

$\Gamma \vdash tlift\ (MReactT\ TI\ TO\ SM)\ t : TMonadic\ (MReactT\ TI\ TO\ SM)\ T$

| *T_Elevate* : $\forall\ \Gamma\ SM\ SM'\ t\ T$,

$smo\_less\_permissive\ SM\ SM' \rightarrow$

$\Gamma \vdash t : TMonadic\ (MNonReact\ SM)\ T \rightarrow$

$\Gamma \vdash televate\ SM'\ t : TMonadic\ (MNonReact\ SM')\ T$

| *T_Get* : $\forall\ \Gamma\ SM\ E\ T$,

$Eff\_lt\ EffR\ E \rightarrow$

$\Gamma \vdash tget\ (MStateT\ T\ E\ SM) : TMonadic\ (MNonReact\ (MStateT\ T\ E\ SM))\ T$

| *T_Put* : $\forall\ \Gamma\ SM\ E\ t\ T$,

$Eff\_lt\ Eff W\ E \rightarrow$

$\Gamma \vdash t : T \rightarrow$

$\Gamma \vdash tput\ (MStateT\ T\ E\ SM)\ t : TMonadic\ (MNonReact\ (MStateT\ T\ E\ SM))\ TNil$

| $T\_RunSt : \forall\ \Gamma\ SM\ E\ t1\ t2\ TS\ T,$

$\Gamma \vdash t1 : TMonadic\ (MNonReact\ (MStateT\ TS\ E\ SM))\ T \rightarrow$

$\Gamma \vdash t2 : TS \rightarrow$

$\Gamma \vdash trunst\ t1\ t2 : TMonadic\ (MNonReact\ SM)\ (TProd\ T\ TS)$

| $T\_RunId : \forall\ \Gamma\ t\ T,$

$\Gamma \vdash t : TMonadic\ (MNonReact\ MIdentity)\ T \rightarrow$

$\Gamma \vdash trunid\ t : T$

| $T\_Pause : \forall\ \Gamma\ SM\ TI\ TO\ T\ t,$

$\Gamma \vdash t : TMonadic\ (MNonReact\ SM)\ (TProd\ TO\ (TArrow\ TI\ (TMonadic\ (MReactT\ TI\ TO\ SM)\ T))) \rightarrow$

$\Gamma \vdash tpause\ (MReactT\ TI\ TO\ SM)\ T\ t : TMonadic\ (MReactT\ TI\ TO\ SM)\ T$

| $T\_Unfold : \forall\ \Gamma\ SM\ TI\ TO\ TA\ TB\ t1\ t2,$

$\Gamma \vdash t1 : TB \rightarrow$

$\Gamma \vdash t2 : TArrow\ TB\ (TMonadic\ (MNonReact\ SM)\ (TSum\ TA\ (TProd\ TO\ (TArrow\ TI\ TB)))) \rightarrow$

$\Gamma \vdash tunfold\ (MReactT\ TI\ TO\ SM)\ TA\ TB\ t1\ t2 : TMonadic\ (MReactT\ TI\ TO\ SM)\ TA$

| $T\_RunRe : \forall\ \Gamma\ SM\ TI\ TO\ T\ t,$

$\Gamma \vdash t : TMonadic\ (MReactT\ TI\ TO\ SM)\ T \rightarrow$

$\Gamma \vdash trunre\ T\ t : TMonadic\ (MNonReact\ SM)\ (TSum\ T\ (TProd\ TO\ (TArrow\ TI\ (TMonadic\ (MReactT\ TI\ TO\ SM)\ T))))$

```
where "Gamma '⊢' t ':' T" := (has_type Γ t T).

Lemma type_unique : ∀ t Γ T1 T2,
      Γ ⊢ t : T1 →
      Γ ⊢ t : T2 → T1 = T2.
```

B.3.2 For configurations

```
Inductive store_matches_mo : store → Mo → Prop :=
  | matches_mo_id : store_matches_mo nil (MNonReact MIdentity)
  | matches_mo_statet : ∀ SM E T t Sto,
                        {} ⊢ t : T →
                        store_matches_mo Sto (MNonReact SM) →
                          store_matches_mo (t::Sto) (MNonReact (MStateT T E
SM))
  | matches_mo_reactt : ∀ SM TI TO Sto,
                        store_matches_mo Sto (MNonReact SM) →
                          store_matches_mo Sto (MReactT TI TO SM).

Reserved Notation "co '|>' T" (at level 40).

Inductive store_all_values : store → Prop :=
  | sav_empty : store_all_values nil
  | sav_cons : ∀ s Sto, value s → store_all_values Sto → store_all_values (s::Sto).

Inductive configuration_has_type : configuration → Ty → Prop :=
  | configuration_has_type_intro : ∀ t T M Sto,
                                   {} ⊢ t : TMonadic M T →
                                   store_all_values Sto →
                                   store_matches_mo Sto M →
                                     (t,Sto) |> TMonadic M T
```

`where` "co |> T" := (*configuration_has_type co T*).

`Lemma` *less_permissive_store_matches* : ∀ *SM1 SM2 Sto*,

smo_less_permissive SM1 SM2 →

store_matches_mo Sto (*MNonReact SM1*) →

store_matches_mo Sto (*MNonReact SM2*).

`Lemma` *more_permissive_store_matches* : ∀ *SM1 SM2 Sto*,

smo_less_permissive SM2 SM1 →

store_matches_mo Sto (*MNonReact SM1*) →

store_matches_mo Sto (*MNonReact SM2*).

`Theorem` *typable_empty_everywhere*:

∀ Γ *T t*,

{} ⊢ *t* : *T* →

Γ ⊢ *t* : *T*.

B.4 Canonical Forms

`Lemma` *canonical_forms_fun* : ∀ *t T1 T2*,

{} ⊢ *t* : (*TArrow T1 T2*) →

value t →

∃ *x u*, *t* = *tabs x T1 u*.

`Lemma` *canonical_forms_pair* : ∀ *t T1 T2*,

{} ⊢ *t* : (*TProd T1 T2*) →

value t →

∃ *t1 t2*, *t = tpair t1 t2*.

`Lemma` *canonical_forms_copair* : ∀ *t T1 T2*,

{} ⊢ *t* : (*TSum T1 T2*) →

value t →

(∃ *t'*, *t = tinl T2 t'*) ∨ (∃ *t'*, *t = tinr T1 t'*).

`Lemma` *canonical_forms_tunit* : ∀ *t*,

{} ⊢ *t* : *TNil* →

value t →

t = tunit.

B.5 Substitution

`Instance` *IdDec* : *@EqDec id eq eq_equivalence*.

Now we define a function FVs that returns any variables that occur free in a term .

`Function` *FVs* (*t* : *tm*) {`struct` *t*} : *list id* :=

`match` *t* `with`

| (*tvar x*) ⇒ (*x* :: *nil*)

| *tapp t1 t2* ⇒ (*FVs t1*) ++ (*FVs t2*)

| *tabs x T t'* ⇒ (*FVs t'*)/{*x*}

| *tunit* ⇒ (*@nil id*)

| *tpair t1 t2* ⇒ (*FVs t1*) ++ (*FVs t2*)

| *tproj t1 t2* ⇒ (*FVs t1*) ++ (*FVs t2*)

| *tinl T t1* ⇒ (*FVs t1*)

| *tinr T t1* ⇒ (*FVs t1*)

| *tcase t1 t2 t3* ⇒ (*FVs t1*) ++ (*FVs t2*) ++ (*FVs t3*)

| *treturn M t1* ⇒ (*FVs t1*)

```
 | tbind T t1 t2 ⇒ (FVs t1) ++ (FVs t2)
 | tlift M t1 ⇒ (FVs t1)
 | televate SM t1 ⇒ (FVs t1)
 | tget SM ⇒ (@nil id)
 | tput SM t1 ⇒ (FVs t1)
 | trunst t1 t2 ⇒ (FVs t1) ++ (FVs t2)
 | trunid t1 ⇒ (FVs t1)
 | tpause M T t1 ⇒ (FVs t1)
 | tunfold M TA TB t1 t2 ⇒ (FVs t1) ++ (FVs t2)
 | trunre T t1 ⇒ (FVs t1)
 end.

Theorem fv_dec : ∀ x t, {In x (FVs t)} + {~ In x (FVs t)}.

Theorem eq_tm_dec : ∀ t1 t2 : tm, {t1 = t2} + {t1 ≠ t2}.

Function convert (n : id) : nat :=
  match n with
  | (Id m) ⇒ m
  end.

Lemma convert_unique_aux :
  ∀ n : id,
  ∃ x : nat,
      convert n = x ∧
      (∀ x' : nat, convert n = x' → x = x').

Lemma convert_unique : ∀ n : id,
       ∃ ! (m : nat), convert n = m.

Definition convert_list := fun l ⇒ map (convert) l.

Definition find_new_id (l : list id) : nat :=
```

```
  let l := convert_list l in
   (1 + (fold_right (fun n x ⇒ max n x) 0 l)).

Lemma find_new_nonzero : ∀ l, find_new_id l ≠ 0.

Definition find_max_id (l : list id) : nat :=
  let l := convert_list l in
  fold_right (fun n x ⇒ max n x) 0 l.

Lemma find_new_max : ∀ l x,
      In x l → (convert x ≤ find_max_id l).

Lemma find_new_id_new : ∀ l,
      let k := (Id (find_new_id l)) in ¬ In k l.
```

B.6 Substitution

```
Reserved Notation "'[' x ':=' s ']' t" (at level 20).

Fixpoint subst (x:id) (s:tm) (t:tm) : tm :=
  match t with
  | tvar x' ⇒
      if eq_id_dec x x' then s else t
  | tapp t1 t2 ⇒
      tapp ([x:=s] t1) ([x:=s] t2)
  | tabs x' T t1 ⇒
      tabs x' T (if eq_id_dec x x' then t1 else ([x:=s] t1))
  | tunit ⇒
      tunit
  | tpair t1 t2 ⇒
      tpair ([x:=s] t1) ([x:=s] t2)
  | tproj t1 t2 ⇒
```

```
    tproj ([x:=s] t1) ([x:=s] t2)
| tinl T t1 ⇒
    tinl T ([x:=s] t1)
| tinr T t1 ⇒
    tinr T ([x:=s] t1)
| tcase t1 t2 t3 ⇒
    tcase ([x:=s] t1) ([x:=s] t2) ([x:=s] t3)
| treturn M t1 ⇒
    treturn M ([x:=s] t1)
| tbind T t1 t2 ⇒
    tbind T ([x:=s] t1) ([x:=s] t2)
| tlift M t1 ⇒
    tlift M ([x:=s] t1)
| televate SM t1 ⇒
    televate SM ([x:=s] t1)
| tget SM ⇒
    tget SM
| tput SM t1 ⇒
    tput SM ([x:=s] t1)
| trunst t1 t2 ⇒
    trunst ([x:=s] t1) ([x:=s] t2)
| trunid t1 ⇒
    trunid ([x:=s] t1)
| tpause M T t1 ⇒
    tpause M T ([x:=s] t1)
| tunfold M TA TB t1 t2 ⇒
    tunfold M TA TB ([x:=s] t1) ([x:=s] t2)
```

| *trunre T t1* ⇒

trunre T ([*x*:=*s*] *t1*)

end

where "'[' x ':=' s ']' t" := (subst *x s t*).

A variable *x* appears free in a term *t* .

Inductive *appears_free_in* : *id* → *tm* → Prop :=

| *afi_var* : ∀ *x*,

appears_free_in x (*tvar x*)

| *afi_app1* : ∀ *x t1 t2*,

appears_free_in x t1 →

appears_free_in x (*tapp t1 t2*)

| *afi_app2* : ∀ *x t1 t2*,

appears_free_in x t2 →

appears_free_in x (*tapp t1 t2*)

| *afi_abs* : ∀ *x y T11 t12*,

y ≠ *x* →

appears_free_in x t12 →

appears_free_in x (*tabs y T11 t12*)

| *afi_pair1* : ∀ *x t1 t2*,

appears_free_in x t1 →

appears_free_in x (*tpair t1 t2*)

| *afi_pair2* : ∀ *x t1 t2*,

appears_free_in x t2 →

appears_free_in x (*tpair t1 t2*)

| *afi_proj1* : ∀ *x t1 t2*,

$appears\_free\_in\ x\ t1 \to$

$appears\_free\_in\ x\ (tproj\ t1\ t2)$

$\mid afi\_proj2 : \forall\ x\ t1\ t2,$

$appears\_free\_in\ x\ t2 \to$

$appears\_free\_in\ x\ (tproj\ t1\ t2)$

$\mid afi\_tinl : \forall\ x\ t\ T,$

$appears\_free\_in\ x\ t \to$

$appears\_free\_in\ x\ (tinl\ T\ t)$

$\mid afi\_tinr : \forall\ x\ t\ T,$

$appears\_free\_in\ x\ t \to$

$appears\_free\_in\ x\ (tinr\ T\ t)$

$\mid afi\_tcase1 : \forall\ x\ t1\ t2\ t3,$

$appears\_free\_in\ x\ t1 \to$

$appears\_free\_in\ x\ (tcase\ t1\ t2\ t3)$

$\mid afi\_tcase2 : \forall\ x\ t1\ t2\ t3,$

$appears\_free\_in\ x\ t2 \to$

$appears\_free\_in\ x\ (tcase\ t1\ t2\ t3)$

$\mid afi\_tcase3 : \forall\ x\ t1\ t2\ t3,$

$appears\_free\_in\ x\ t3 \to$

$appears\_free\_in\ x\ (tcase\ t1\ t2\ t3)$

$\mid afi\_treturn : \forall\ x\ t\ M,$

$appears\_free\_in\ x\ t \to$

$appears\_free\_in\ x\ (treturn\ M\ t)$

$\mid afi\_tbind1 : \forall\ x\ T\ t1\ t2,$

$appears\_free\_in\ x\ t1 \to$

$appears\_free\_in\ x\ (tbind\ T\ t1\ t2)$

| $afi\_tbind2 : \forall\ x\ T\ t1\ t2,$

$appears\_free\_in\ x\ t2 \rightarrow$

$appears\_free\_in\ x\ (tbind\ T\ t1\ t2)$

| $afi\_tlift : \forall\ x\ t\ M,$

$appears\_free\_in\ x\ t \rightarrow$

$appears\_free\_in\ x\ (tlift\ M\ t)$

| $afi\_televate : \forall\ x\ t\ M,$

$appears\_free\_in\ x\ t \rightarrow$

$appears\_free\_in\ x\ (televate\ M\ t)$

| $afi\_tput : \forall\ x\ t\ M,$

$appears\_free\_in\ x\ t \rightarrow$

$appears\_free\_in\ x\ (tput\ M\ t)$

| $afi\_trunst1 : \forall\ x\ t1\ t2,$

$appears\_free\_in\ x\ t1 \rightarrow$

$appears\_free\_in\ x\ (trunst\ t1\ t2)$

| $afi\_trunst2 : \forall\ x\ t1\ t2,$

$appears\_free\_in\ x\ t2 \rightarrow$

$appears\_free\_in\ x\ (trunst\ t1\ t2)$

| $afi\_trunid : \forall\ x\ t,$

$appears\_free\_in\ x\ t \rightarrow$

$appears\_free\_in\ x\ (trunid\ t)$

| $afi\_tpause : \forall\ x\ M\ T\ t,$

$appears\_free\_in\ x\ t \rightarrow$

$appears\_free\_in\ x\ (tpause\ M\ T\ t)$

```
  | afi_tunfold1 : ∀ x M TA TB t1 t2,
      appears_free_in x t1 →
      appears_free_in x (tunfold M TA TB t1 t2)
  | afi_tunfold2 : ∀ x M TA TB t1 t2,
      appears_free_in x t2 →
      appears_free_in x (tunfold M TA TB t1 t2)
  | afi_trunre : ∀ x T t,
      appears_free_in x t →
      appears_free_in x (trunre T t)
.

Lemma afi_dec : ∀ x t, appears_free_in x t ∨ ¬ appears_free_in x t.

Lemma FVs_AFI_eq : ∀ (x:id) (t:tm), In x (FVs t) ↔ appears_free_in x t.

Definition closed (t:tm) := ∀ x, ¬ appears_free_in x t.

Lemma context_invariance : ∀ Gamma Gamma' t S,
      Gamma |- t \in S →
      (∀ x, appears_free_in x t → Gamma x = Gamma' x) →
      Gamma' |- t \in S.

Lemma free_in_context : ∀ x t T Gamma,
      appears_free_in x t →
        Gamma |- t \in T →
        (∃ T', Gamma x = Some T').

Lemma subst_preserves_typing : ∀ Gamma x U v t T,
      (extend Gamma x U) |- t \in T →
                        \empty |- v \in U →
                          Gamma |- ([x:=v] t) \in T.

Corollary typable_empty_closed : ∀ t T,
```

```
      \empty |- t \in T →
      closed t.

Lemma vacuous_substitution : ∀ t x,
     ¬ appears_free_in x t →
     ∀ t', [x:=t']t = t.

Lemma subst_closed: ∀ t,
      closed t →
      ∀ x t', [x:=t']t = t.

Lemma subst_not_afi : ∀ t x v,
      closed v →
      ¬ appears_free_in x ([x:=v]t).

Lemma duplicate_subst : ∀ t' x t v,
      closed v → [x:=t]([x:=v]t') = [x:=v]t'.

Lemma swap_subst : ∀ t x x1 v v1,
        x ≠ x1 →
       closed v →
      closed v1 →
      [x1:=v1]([x:=v]t) = [x:=v]([x1:=v1]t).

Lemma subst_rewrite : ∀ (x : id) (t t' : tm) (T : Ty),
      \empty |- t \in T →
      ([x:=t']t = t).

Lemma typable_empty_config : ∀ t Sto T,
      (t,Sto) |> T → \empty |- t \in T.
```

B.7 Reduction

B.7.1 Lambda-calculus and monadic reduction relations

```
Reserved Notation "t1 '~~>' t2" (at level 40).

Reserved Notation "t1 '~~~>' t2" (at level 40).

Inductive step : tm → tm → Prop :=
  | ST_AppAbs : ∀ x T t12 v2,
                value v2 →
                (tapp (tabs x T t12) v2) ~~> [x:=v2]t12
  | ST_App1 : ∀ t1 t1' t2,
                t1 ~~> t1' →
                tapp t1 t2 ~~> tapp t1' t2
  | ST_App2 : ∀ v1 t2 t2',
                value v1 →
                t2 ~~> t2' →
                tapp v1 t2 ~~> tapp v1 t2'
  | ST_Pair1 : ∀ t1 t1' t2,
                t1 ~~> t1' →
                tpair t1 t2 ~~> tpair t1' t2
  | ST_Pair2 : ∀ v1 t2 t2',
                value v1 →
                t2 ~~> t2' →
                tpair v1 t2 ~~> tpair v1 t2'
  | ST_Proj1 : ∀ t1 t1' t2,
                t1 ~~> t1' →
                tproj t1 t2 ~~> tproj t1' t2
  | ST_Proj2 : ∀ v1 t2 t2',
```

value v1 →

t2 ~~> *t2'* →

tproj v1 t2 ~~> *tproj v1 t2'*

| *ST_Proj* : ∀ *v1 v2 v3*,

value v1 →

value v2 →

value v3 →

tproj (*tpair v1 v2*) *v3* ~~> *tapp* (*tapp v3 v1*) *v2*

| *ST_InL* : ∀ *T t1 t1'*,

t1 ~~> *t1'* →

tinl T t1 ~~> *tinl T t1'*

| *ST_InR* : ∀ *T t1 t1'*,

t1 ~~> *t1'* →

tinr T t1 ~~> *tinr T t1'*

| *ST_Case* : ∀ *t1 t1' t2 t3*,

t1 ~~> *t1'* →

tcase t1 t2 t3 ~~> *tcase t1' t2 t3*

| *ST_CaseL* : ∀ *v1 T t2 t3*,

value v1 →

tcase (*tinl T v1*) *t2 t3* ~~> *tapp t2 v1*

| *ST_CaseR* : ∀ *v1 T t2 t3*,

value v1 →

tcase (*tinr T v1*) *t2 t3* ~~> *tapp t3 v1*

| *ST_Return* : ∀ *M t1 t1'*,

t1 ~~> *t1'* →

treturn M t1 ~~> *treturn M t1'*

| *ST_Bind1* : ∀ *T t1 t1' t2*,

$t1 \tilde{\tilde{}}> t1' \rightarrow$

$tbind\ T\ t1\ t2 \tilde{\tilde{}}> tbind\ T\ t1'\ t2$

| $ST\_Bind2 : \forall\ T\ t1\ t2\ t2'$,

$value\ t1 \rightarrow$

$t2 \tilde{\tilde{}}> t2' \rightarrow$

$tbind\ T\ t1\ t2 \tilde{\tilde{}}> tbind\ T\ t1\ t2'$

| $ST\_Lift : \forall\ M\ t1\ t1'$,

$t1 \tilde{\tilde{}}> t1' \rightarrow$

$tlift\ M\ t1 \tilde{\tilde{}}> tlift\ M\ t1'$

| $ST\_Elevate : \forall\ SM\ t1\ t1'$,

$t1 \tilde{\tilde{}}> t1' \rightarrow$

$televate\ SM\ t1 \tilde{\tilde{}}> televate\ SM\ t1'$

| $ST\_Put : \forall\ SM\ t1\ t1'$,

$t1 \tilde{\tilde{}}> t1' \rightarrow$

$tput\ SM\ t1 \tilde{\tilde{}}> tput\ SM\ t1'$

| $ST\_RunSt1 : \forall\ t1\ t1'\ t2$,

$t1 \tilde{\tilde{}}> t1' \rightarrow$

$trunst\ t1\ t2 \tilde{\tilde{}}> trunst\ t1'\ t2$

| $ST\_RunSt2 : \forall\ t1\ t2\ t2'$,

$value\ t1 \rightarrow$

$t2 \tilde{\tilde{}}> t2' \rightarrow$

$trunst\ t1\ t2 \tilde{\tilde{}}> trunst\ t1\ t2'$

| $ST\_RunId : \forall\ t1\ t1'$,

$t1 \tilde{\tilde{}}> t1' \rightarrow$

$trunid\ t1 \tilde{\tilde{}}> trunid\ t1'$

| $ST\_RunIdMo : \forall\ t1\ t1'$,

$value\ t1 \rightarrow$

```
                                (t1,nil) ~~~> (t1',nil) →
                                trunid t1 ~~> trunid t1'
  | ST_RunIdRet : ∀ M v,
                                value v →
                                trunid (treturn M v) ~~> v
  | ST_Pause : ∀ M T t1 t1',
                                t1 ~~> t1' →
                                tpause M T t1 ~~> tpause M T t1'
  | ST_Unfold1 : ∀ M TA TB t1 t1' t2,
                                t1 ~~> t1' →
                                tunfold M TA TB t1 t2 ~~> tunfold M TA TB t1' t2
  | ST_Unfold2 : ∀ M TA TB t1 t2 t2',
                                value t1 →
                                t2 ~~> t2' →
                                tunfold M TA TB t1 t2 ~~> tunfold M TA TB t1 t2'
  | ST_RunRe : ∀ T t1 t1',
                                t1 ~~> t1' →
                                trunre T t1 ~~> trunre T t1'

with step_mo : configuration → configuration → Prop :=
  | STM_LC : ∀ t t' Sto,
                                  t ~~> t' →
                                  (t,Sto) ~~~> (t',Sto)
  | STM_Bind1 : ∀ T t1 t1' t2 Sto Sto',
                                  value t1 →
                                  value t2 →
                                  (t1,Sto) ~~~> (t1',Sto') →
```

(*tbind T t1 t2,Sto*) ~~~> (*tbind T t1' t2,Sto'*)

| *STM_BindRet* : ∀ *v1 v2 T M Sto*,

value v1 →

value v2 →

(*tbind T* (*treturn M v1*) *v2,Sto*) ~~~> (*tapp v2 v1,Sto*)

| *STM_LiftSt* : ∀ *t t' Sto Sto' s TS b SM*,

value t →

(*t,Sto*) ~~~> (*t',Sto'*) →

(*tlift* (*MNonReact* (*MStateT TS b SM*)) *t,s*::*Sto*) ~~~>

(*tlift* (*MNonReact* (*MStateT TS b SM*)) *t',s*::*Sto'*)

| *STM_LiftRe* : ∀ *t t' Sto Sto' TI TO SM*,

value t →

(*t,Sto*) ~~~> (*t',Sto'*) →

(*tlift* (*MReactT TI TO SM*) *t,Sto*) ~~~>

(*tlift* (*MReactT TI TO SM*) *t',Sto'*)

| *STM_LiftRetSt* : ∀ *v TS b SM Sto*,

value v →

(*tlift* (*MNonReact* (*MStateT TS b SM*)) (*treturn* (*MNonReact SM*) *v*),*Sto*)

~~~> (*treturn* (*MNonReact* (*MStateT TS b SM*)) *v,Sto*)

| *STM_LiftRetRe* : ∀ *v TI TO SM Sto*,

value v →

(*tlift* (*MReactT TI TO SM*) (*treturn* (*MNonReact SM*) *v*),*Sto*)

~~~> (*treturn* (*MReactT TI TO SM*) *v,Sto*)

| *STM_Get* : ∀ *SM Sto s*,

(*tget SM,s*::*Sto*)

~~~> (*treturn* (*MNonReact SM*) *s,s*::*Sto*)
~~~

| *STM_Put* : ∀ *v SM Sto s*,

value v →

(*tput SM v*,*s*::*Sto*)

~~~> (*treturn* (*MNonReact SM*) *tunit*,*v*::*Sto*)

| *STM_Elevate* : ∀ *SM t1 t1' Sto Sto'*,

value t1 →

(*t1*,*Sto*) ~~~> (*t1'*,*Sto'*) →

(*televate SM t1*,*Sto*)

~~~> (*televate SM t1'*,*Sto'*)

| *STM_ElevateRet* : ∀ *SM SM' v Sto*,

value v →

(*televate SM'* (*treturn* (*MNonReact SM*) *v*),*Sto*)

~~~> (*treturn* (*MNonReact SM'*) *v*,*Sto*)

| *STM_RunSt* : ∀ *t1 t1' s s' Sto Sto'*,

value t1 →

value s →

(*t1*,*s*::*Sto*) ~~~> (*t1'*,*s'*::*Sto'*) →

(*trunst t1 s*,*Sto*) ~~~> (*trunst t1' s'*,*Sto'*)

| *STM_RunStRet* : ∀ *t1 s Sto TS b SM*,

value t1 →

value s →

(*trunst* (*treturn* (*MNonReact* (*MStateT TS b SM*)) *t1*) *s*,*Sto*)

~~~> (*treturn* (*MNonReact SM*) (*tpair t1 s*),*Sto*)

| *STM_Unfold* : ∀ *t1 t2 Sto TI TO SM TA TB*,

value t1 →

value t2 →

(*tunfold* (*MReactT TI TO SM*) *TA TB t1 t2*,*Sto*) ~~~>

(*tbind TA*

(*tlift* (*MReactT TI TO SM*) (*tapp t2 t1*))

(*tabs* 0 (*TSum TA* (*TProd TO* (*TArrow TI TB*)))

(*tcase* (*tvar* 0)

(*tabs* 1 *TA* (*treturn* (*MReactT TI TO SM*) (*tvar*

1)))

(*tabs* 1 (*TProd TO* (*TArrow TI TB*))

(*tproj*

(*tvar* 1)

(*tabs* 2 *TO*

(*tabs* 3 (*TArrow TI TB*)

(*tpause* (*MReactT TI TO SM*) *TA*

(*treturn* (*MNonReact SM*)

(*tpair* (*tvar* 2)

(*tabs* 4 *TI*

(*tunfold*

(*MReactT TI TO SM*) *TA TB*

(*tapp* (*tvar*

3) (*tvar* 4))

t2))))))))))),*Sto*)

| *STM_PauseBind* : ∀ *t1 t2 Sto TI TO SM T1 T2*,

value t1 →

value t2 →

(*tbind*

T2

(*tpause*

(*MReactT TI TO SM*)

T1

t1)

t2,*Sto*) ~~~>

(*tpause*

(*MReactT TI TO SM*)

T2

(*tbind*

(*TProd TO* (*TArrow TI* (*TMonadic* (*MReactT TI TO SM*) *T2*)))

t1

(*tabs* 0 (*TProd TO* (*TArrow TI* (*TMonadic* (*MReactT TI TO SM*) *T1*)))

(*tproj*

(*tvar* 0)

(*tabs* 1 *TO*

(*tabs* 2 (*TArrow TI* (*TMonadic* (*MReactT TI TO SM*) *T1*))

(*treturn* (*MNonReact SM*)

(*tpair*

(*tvar* 1)

(*tabs* 3 *TI*

(*tbind*

T2

(*tapp* (*tvar* 2) (*tvar* 3))

t2))))))))),*Sto*)

| *STM_RunRe* : ∀ *T t1 t1' Sto Sto'*,

value t1 →

```
                    (t1,Sto) ~~~> (t1',Sto') →
                    (trunre T t1,Sto) ~~~> (trunre T t1',Sto')
  | STM_RunReRet : ∀ v TI TO SM TA Sto,
                    value v →
                    (trunre TA (treturn (MReactT TI TO SM) v),Sto) ~~~>
                      (treturn
                        (MNonReact SM)
                        (tinl (TProd TO (TArrow TI (TMonadic (MReactT TI TO
SM) TA))) v),Sto)
  | STM_RunRePause : ∀ v TI TO SM TA Sto,
                    value v →
                    (trunre TA (tpause (MReactT TI TO SM) TA v),Sto) ~~~>
                      (tbind (TSum TA (TProd TO (TArrow TI (TMonadic (MRe-
actT TI TO SM) TA))))
                             v
                             (tabs 0 (TProd TO (TArrow TI (TMonadic (MRe-
actT TI TO SM) TA)))
                                     (treturn (MNonReact SM) (tinr TA (tvar
0)))),Sto)
where "k1 '~~>' k2" := (step k1 k2)
                    and "k1 '~~~>' k2" := (step_mo k1 k2).

Section Reflexive_Transitive_Closure.
Variables (X : Type) (R : relation X).

Inductive rt_closure : relation X :=
    | rtc_refl : ∀ (x : X),
                    rt_closure x x
    | rtc_step : ∀ (x y z : X),
```

$R\ x\ y \rightarrow$

$rt\_closure\ y\ z \rightarrow$

$rt\_closure\ x\ z.$

Theorem $rtc\_R' : \forall\ (x\ y : X),$

$R\ x\ y \rightarrow rt\_closure\ x\ y.$

Theorem $rtc\_Trans' : \forall\ (x\ y\ z : X),$

$rt\_closure\ x\ y \rightarrow$

$rt\_closure\ y\ z \rightarrow$

$rt\_closure\ x\ z.$

Lemma $rtc\_Last : \forall\ (x\ y\ z : X),$

$rt\_closure\ x\ y \rightarrow R\ y\ z \rightarrow rt\_closure\ x\ z.$

B.7.2 Induction Principles

Lemma $rtc\_ind\_with\_trans : \forall\ (P : X \rightarrow X \rightarrow$ Prop),

$(\forall\ x : X, P\ x\ x) \rightarrow$

$(\forall\ x\ y : X, R\ x\ y \rightarrow P\ x\ y) \rightarrow$

$(\forall\ y\ x\ z : X, rt\_closure\ x\ y \rightarrow P\ x\ y \rightarrow rt\_closure\ y\ z \rightarrow P\ y\ z \rightarrow P\ x\ z) \rightarrow$

$\forall\ x\ y : X, rt\_closure\ x\ y \rightarrow P\ x\ y.$

Lemma $rtc\_ind\_steps\_last : \forall\ (P : X \rightarrow X \rightarrow$ Prop),

$(\forall\ x : X, P\ x\ x) \rightarrow$

$(\forall\ y\ x\ z : X, rt\_closure\ x\ y \rightarrow P\ x\ y \rightarrow R\ y\ z \rightarrow P\ x\ z) \rightarrow$

$\forall\ x\ y : X, rt\_closure\ x\ y \rightarrow P\ x\ y.$

End *Reflexive_Transitive_Closure.*

Add *Parametric Relation T R : T (@rt_closure T R)*

reflexivity *proved* by *(@rtc_refl _ _)*

transitivity *proved* by *(@rtc_Trans' _ _)*

as *RTC_Rel*.

Notation *multistep* := (*rt_closure tm step*).

Notation "t1 '~~>*' t2" := (*multistep t1 t2*) (at level 40).

Notation *multimostep* := (*rt_closure configuration step_mo*).

Notation "t '~~~>*' t'" := (*multimostep t t'*) (at level 40).

Tactic Notation "RTClosure_Ind_1" constr(*name*) := *Case_aux RTClosure_Ind_1 name*.

Tactic Notation "RTClosure_Ind_2" constr(*name*) := *Case_aux RTClosure_Ind_2 name*.

Tactic Notation "RTClosure_Ind_3" constr(*name*) := *Case_aux RTClosure_Ind_3 name*.

Two tactics for using *rtc_ind_with_trans* and *rtc_ind_steps_last*

Tactic Notation "R*" "induction" *hyp*(*H*) "using" "trans" :=

(induction *H* using *rtc_ind_with_trans*; [*Case_aux RTClosure_Ind_1* "Base"

|*Case_aux RTClosure_Ind_2* "Step"

|*Case_aux RTClosure_Ind_3* "Transitivity"]).

Tactic Notation "R*" "induction" *hyp*(*H*) "using" "steps" :=

(induction *H* using *rtc_ind_steps_last*; [*Case_aux RTClosure_Ind_1* "Base"

|*Case_aux RTClosure_Ind_2* "Step"]).

Theorem *rtc_mstep_refl* : ∀ (*t*:*tm*),

t ~~>* *t*.

Theorem *rtc_mstep_refl_mo* : ∀ (*co*:*configuration*),

co ~~~>* *co*.

Theorem *rtc_R* : ∀ (*t*:*tm*) (*t'*:*tm*),

t ~~> *t'* →

t ~~>* *t'*.

Theorem *rtc_R_mo* : ∀ *co co'*,

co ~~~> *co'* →

co ~~~>* *co'*.

Theorem *rtc_Trans* : ∀ (*t*:*tm*) (*t'*:*tm*) (*t''*:*tm*),

t ~~>* *t'* →

t' ~~>* *t''* →

t ~~>* *t''*.

Theorem *rtc_Trans_mo* : ∀ *co co' co''*,

co ~~~>* *co'* →

co' ~~~>* *co''* →

co ~~~>* *co''*.

Inductive *same_length* {*A*:Type} : *list A* → *list A* → Prop :=

| *same_length_nil* : *same_length nil nil*

| *same_length_cons* : ∀ *x y l1 l2*,

same_length l1 l2 → *same_length* (*x*::*l1*) (*y*::*l2*).

Lemma *same_length_refl* : ∀ {*A*:Type} (*l*:*list A*), *same_length l l*.

Lemma *same_length_trans* : ∀ {*A*:Type} (*l1 l2 l3*:*list A*), *same_length l1 l2* → *same_length l2 l3* → *same_length l1 l3*.

Lemma *step_mo_same_length* : ∀ *t t' Sto Sto'*,

(*t*,*Sto*) ~~~> (*t'*,*Sto'*) →

same_length Sto Sto'.

Lemma *step_mo_same_length_star* : ∀ *t t' Sto Sto'*,

(*t*,*Sto*) ~~~>* (*t'*,*Sto'*) →

same_length Sto Sto'.

Some lemmas about values and done configurations. Lemma *value_not_step* : ∀ *v*, *value v* → ∀ *t*, ~(*v* ~~> *t*).

Lemma *step_not_value* : ∀ *t*, (∃ *t'*, *t* ~~> *t'*) → ¬ *value t.*

Theorem *done_not_step* : ∀ *co*, *done_mo co* → ∀ *co'*, ¬*co* ~~~> *co'*.

Determinism

Theorem *step_deterministic'* : ∀ *t u v*,

((*t* ~~> *u* → *t* ~~> *v* → *u* = *v*)

∧ ∀ *Sto Sto' Sto''*,

((*t*,*Sto*) ~~~> (*u*,*Sto'*) →

(*t*,*Sto*) ~~~> (*v*,*Sto''*) →

(*u* = *v* ∧ *Sto'* = *Sto''*))).

Theorem *step_deterministic* : ∀ *t u v*, *t* ~~> *u* → *t* ~~> *v* → *u* = *v*.

Theorem *step_deterministic_mo* : ∀ *co1 co2 co3*, *co1* ~~~> *co2* → *co1* ~~~> *co3* → *co2* = *co3.*

Lemmas concerning how values/done congfigurations

interact with the step and multistep reduction relations.

Lemma *step_monad_second* : ∀ *t t' t'' Sto Sto'*,

t ~~> *t'* →

value t'' →

(*t*,*Sto*) ~~~>* (*t''*,*Sto'*) →

(*t'*,*Sto*) ~~~>* (*t''*,*Sto'*).

Lemma *step_value_second* : ∀ *t t' v*,

value v →

t ~~> *t'* →

t ~~>* *v* →

t' ~~>* *v*.

Lemma *value_unique* : ∀ *t v v'*,

value v →

value v' →

t ~~>* *v* →

t ~~>* *v'* →

v = *v'*.

Lemma *step_same_value* : ∀ *t t' v*,

value v →

t ~~>* *t'* →

t ~~>* *v* →

t' ~~>* *v*.

Lemma *step_done_second* : ∀ *co co' co''*,

done_mo co'' →

co ~~~> *co'* →

co ~~~>* *co''* →

co' ~~~>* *co''*.

Lemma *step_same_done* : ∀ *co co' co''*,

done_mo co'' →

co ~~~>* *co'* →

co ~~~>* *co''* →

co' ~~~>* *co''*.

Lemma *done_unique* : ∀ *co co' co''*,

done_mo co' →

done_mo co'' →

co ~~~>* *co'* →

co ~~~>* *co''* →

co' = co''.

Lemma *done_multistep_only_self* : ∀ *co*, *done_mo co* → ∀ *co'*, *co* ~~~>* *co'* → *co = co'*.

Lemma *nonval_step_step_mo* : ∀ *t t' Sto Sto'*,

¬ *value t* →

(*t*,*Sto*) ~~~> (*t'*,*Sto'*) →

(*t* ~~> *t'* ∧ *Sto = Sto'*).

Lemma *step_mo_not_value_step* : ∀ *t t' Sto Sto'*,

(*t*,*Sto*) ~~~> (*t'*,*Sto'*) →

¬ *value t* →

t ~~> *t'*.

Lemma *step_mo_value_not_step* : ∀ *t t' Sto Sto'*,

(*t*,*Sto*) ~~~> (*t'*,*Sto'*) →

value t →

¬ (*t* ~~> *t'*).

Lemma *done_val* : ∀ *t Sto*,

done_mo (*t*,*Sto*) → *value t*.

Lemma *step_mo_pure* : ∀ *t t' Sto t'' Sto'*,

t ~~> *t'* →

(*t*,*Sto*) ~~~> (*t''*,*Sto'*) →

(*t' = t''* ∧ *Sto = Sto'*).

Congruence lemmas on multistep

Lemma *Congruence_App1* : ∀ *t1 t1' t2*,

(*t1* ~~>* *t1'*) →

(*tapp t1 t2*) ~~>* (*tapp t1' t2*).

Lemma *Congruence_App2* : ∀ *v t t'*,

value v →

(*t* ~~>* *t'*) →

(*tapp v t*) ~~>* (*tapp v t'*).

Lemma *Congruence_Pair1* : ∀ *t1 t1' t2*,

(*t1* ~~>* *t1'*) →

(*tpair t1 t2*) ~~>* (*tpair t1' t2*).

Lemma *Congruence_Pair2* : ∀ *v1 t2 t2'*,

value v1 → *t2* ~~>* *t2'* → (*tpair v1 t2*) ~~>* (*tpair v1 t2'*).

Lemma *Congruence_Proj1* : ∀ *t1 t1' t2*,

(*t1* ~~>* *t1'*) →

(*tproj t1 t2*) ~~>* (*tproj t1' t2*).

Lemma *Congruence_Proj2* : ∀ *v1 t2 t2'*,

value v1 → *t2* ~~>* *t2'* → (*tproj v1 t2*) ~~>* (*tproj v1 t2'*).

Lemma *Congruence_Tinl* : ∀ *T t1 t2*,

t1 ~~>* *t2* → (*tinl T t1*) ~~>* (*tinl T t2*).

Lemma *Congruence_Tinr* : ∀ *T t1 t2*,

t1 ~~>* *t2* → (*tinr T t1*) ~~>* (*tinr T t2*).

Lemma *Congruence_Tcase* : ∀ *t1 t1' t2 t3*,

t1 ~~>* *t1'* → *tcase t1 t2 t3* ~~>* *tcase t1' t2 t3*.

Lemma *Congruence_ST_CaseL* : ∀ *T v1 t2 t3*,

value v1 →

tcase (*tinl T v1*) *t2 t3* ~~>* *tapp t2 v1*.

Lemma *Congruence_ST_CaseR* : ∀ *T v1 t2 t3*,

value v1 →

tcase (*tinr T v1*) *t2 t3* ~~>* *tapp t3 v1*.

Lemma *Congruence_LC* : ∀ *t1 t1' Sto*,

t1 ~~>* *t1'* → (*t1*,*Sto*) ~~~>* (*t1'*,*Sto*).

Lemma *step_val_done* : ∀ *t vt Sto t' Sto'*,

t ~~>* *vt* →

value vt →

(*t*,*Sto*) ~~~>* (*t'*,*Sto'*) →

done_mo (*t'*,*Sto'*) →

(*vt*,*Sto*) ~~~>* (*t'*,*Sto'*).

Lemma *Congruence_LC_Ret* : ∀ *M t1 t1'*,

t1 ~~>* *t1'* → *treturn M t1* ~~>* *treturn M t1'*.

Lemma *Congruence_Treturn* : ∀ *t1 t1' M Sto*,

t1 ~~>* *t1'* → (*treturn M t1*,*Sto*) ~~~>* (*treturn M t1'*,*Sto*).

Lemma *Congruence_Tbind1* : ∀ *T t1 t1' t2*,

t1 ~~>* *t1'* →

tbind T t1 t2 ~~>* *tbind T t1' t2*.

Lemma *Congruence_Tbind2* : ∀ *T t1 t2 t2'*,

value t1 →

t2 ~~>* *t2'* →

tbind T t1 t2 ~~>* *tbind T t1 t2'*.

Lemma *Congruence_Mo_Bind1* : ∀ *T t1 Sto t1' Sto' t2*,

value t2 →

(*t1,Sto*) ~~~>* (*t1',Sto'*) →

(*tbind T t1 t2,Sto*) ~~~>* (*tbind T t1' t2,Sto'*).

Lemma *Congruence_BindRet* : ∀ *T* (*v1 v2* : *tm*) (*M* : *Mo*) (*Sto* : *store*),

value v1 →

value v2 →

(*tbind T* (*treturn M v1*) *v2*, *Sto*) ~~~>* (*tapp v2 v1*, *Sto*).

Lemma *Congruence_LiftRetSt* : ∀ *T b M v Sto*,

value v →

(*tlift* (*MNonReact* (*MStateT T b M*)) (*treturn* (*MNonReact M*) *v*),*Sto*) ~~~>* (*treturn* (*MNonReact* (*MStateT T b M*)) *v*,*Sto*).

Lemma *Congruence_LiftRetRe* : ∀ *v TI TO SM Sto*,

value v →

(*tlift* (*MReactT TI TO SM*) (*treturn* (*MNonReact SM*) *v*),*Sto*)

~~~>* (*treturn* (*MReactT TI TO SM*) *v*,*Sto*).

Lemma *Congruence_LiftSt* :

∀ (*t t'* : *tm*) (*Sto Sto'* : *store*) (*s* : *tm*) (*TS* : *Ty*) *b* (*SM* : *SMo*),

value t →

(*t*, *Sto*) ~~~>* (*t'*, *Sto'*) →

(*tlift* (*MNonReact* (*MStateT TS b SM*)) *t*, *s* :: *Sto*) ~~~>*

(*tlift* (*MNonReact* (*MStateT TS b SM*)) *t'*, *s* :: *Sto'*).

Lemma *Congruence_LiftRe* : ∀ *t t' Sto Sto' TI TO SM*,

value t →

(*t*,*Sto*) ~~~>* (*t'*,*Sto'*) →

(*tlift* (*MReactT TI TO SM*) *t*,*Sto*) ~~~>*

(*tlift* (*MReactT TI TO SM*) *t'*,*Sto'*).

Lemma *Congruence_Lift* : ∀ *M t1 t1'*,
~~~

$t1 \tilde{\tilde{}}>^* t1' \rightarrow$

$tlift\ M\ t1 \tilde{\tilde{}}>^* tlift\ M\ t1'.$

Lemma *Congruence_STM_Elevate* : $\forall$ $(SM : SMo)$ $(t\ t' : tm)$ $(Sto\ Sto' : store)$,

$value\ t \rightarrow$

$(t, Sto) \tilde{\tilde{\tilde{}}}>^* (t', Sto') \rightarrow$

$(televate\ SM\ t, Sto) \tilde{\tilde{\tilde{}}}>^* (televate\ SM\ t', Sto').$

Lemma *Congruence_STM_ElevateRet* : $\forall$ $(SM\ SM' : SMo)$ $(v : tm)$ $(Sto : store)$,

$value\ v \rightarrow$

$(televate\ SM'\ (treturn\ (MNonReact\ SM)\ v), Sto) \tilde{\tilde{\tilde{}}}>^*$

$(treturn\ (MNonReact\ SM')\ v, Sto).$

Lemma *Congruence_ST_Elevate* : $\forall$ $(SM : SMo)$ $(t1\ t1' : tm)$,

$t1 \tilde{\tilde{}}>^* t1' \rightarrow$

$televate\ SM\ t1 \tilde{\tilde{}}>^* televate\ SM\ t1'.$

Lemma *Congruence_ST_Put* : $\forall$ $(SM : SMo)$ $(t1\ t1' : tm)$,

$t1 \tilde{\tilde{}}>^* t1' \rightarrow$

$tput\ SM\ t1 \tilde{\tilde{}}>^* tput\ SM\ t1'.$

Lemma *Congruence_PutE* : $\forall$ $SM\ t\ Sto\ t'$,

$t \tilde{\tilde{}}>^* t' \rightarrow$

$(tput\ SM\ t,Sto) \tilde{\tilde{\tilde{}}}>^* (tput\ SM\ t',Sto).$

Lemma *Congruence_RunId* : $\forall$ $t\ t'$,

$t \tilde{\tilde{}}>^* t' \rightarrow$

$trunid\ t \tilde{\tilde{}}>^* trunid\ t'.$

Lemma *Congruence_RunIdMo* : $\forall$ $t\ t'$,

$(t,nil) \tilde{\tilde{\tilde{}}}>^* (t',nil) \rightarrow$

$trunid\ t \tilde{\tilde{}}>^* trunid\ t'.$

Lemma *Congruence_RunSt1* : $\forall$ $t1\ t1'\ t2$,

$t1 \tilde{~}\tilde{~}>^* t1' \rightarrow$

$trunst\ t1\ t2 \tilde{~}\tilde{~}>^* trunst\ t1'\ t2.$

Lemma $Congruence\_RunSt2 : \forall\ t1\ t2\ t2',$

$value\ t1 \rightarrow$

$t2 \tilde{~}\tilde{~}>^* t2' \rightarrow$

$trunst\ t1\ t2 \tilde{~}\tilde{~}>^* trunst\ t1\ t2'.$

Lemma $step\_mo\_still\_values : \forall\ t\ t'\ Sto\ Sto',$

$store\_all\_values\ Sto \rightarrow$

$(t,Sto) \tilde{~}\tilde{~}\tilde{~}> (t',Sto') \rightarrow$

$store\_all\_values\ Sto'.$

Lemma $Congruence\_STM\_RunSt : \forall\ (t\ t'\ s\ s' : tm)\ (Sto\ Sto' : store),$

$value\ s \rightarrow$

$store\_all\_values\ Sto \rightarrow$

$(t,(s :: Sto)) \tilde{~}\tilde{~}\tilde{~}>^* (t',(s' :: Sto')) \rightarrow$

$(trunst\ t\ s, Sto) \tilde{~}\tilde{~}\tilde{~}>^* (trunst\ t'\ s', Sto').$

Lemma $Congruence\_STM\_RunStRet : \forall\ (t1\ s : tm)\ (Sto : store)\ (TS : Ty)\ b$

$(SM : SMo),$

$value\ t1 \rightarrow$

$value\ s \rightarrow$

$(trunst\ (treturn\ (MNonReact\ (MStateT\ TS\ b\ SM))\ t1)\ s, Sto) \tilde{~}\tilde{~}\tilde{~}>^*$

$(treturn\ (MNonReact\ SM)\ (tpair\ t1\ s), Sto).$

Lemma $Congruence\_ST\_Pause : \forall\ M\ T\ t1\ t1',$

$t1 \tilde{~}\tilde{~}>^* t1' \rightarrow$

$tpause\ M\ T\ t1 \tilde{~}\tilde{~}>^* tpause\ M\ T\ t1'.$

Lemma $Congruence\_ST\_RunRe : \forall\ (T : Ty)\ (t1\ t1' : tm),$

$t1 \tilde{~}\tilde{~}>^* t1' \rightarrow trunre\ T\ t1 \tilde{~}\tilde{~}>^* trunre\ T\ t1'.$

```
Lemma Congruence_ST_Unfold1 : ∀ M TA TB t1 t1' t2,
                    t1 ~~>* t1' →
                    tunfold M TA TB t1 t2 ~~>* tunfold M TA TB t1' t2.

Lemma Congruence_ST_Unfold2 : ∀ M TA TB t1 t2 t2',
                    value t1 →
                    t2 ~~>* t2' →
                    tunfold M TA TB t1 t2 ~~>* tunfold M TA TB t1 t2'.

Lemma Congruence_STM_Unfold : ∀ t1 t2 Sto TI TO SM TA TB,
                        value t1 →
                        value t2 →
                        (tunfold (MReactT TI TO SM) TA TB t1 t2,Sto) ~~~>*
                        (tbind TA
                                (tlift (MReactT TI TO SM) (tapp t2 t1))
                                (tabs 0 (TSum TA (TProd TO (TArrow TI TB)))
                                   (tcase (tvar 0)
                                       (tabs 1 TA (treturn (MReactT TI TO SM) (tvar
1)))

                                       (tabs 1 (TProd TO (TArrow TI TB))
                                           (tproj
                                               (tvar 1)
                                               (tabs 2 TO
                                                  (tabs 3 (TArrow TI TB)
                                                      (tpause (MReactT TI TO SM) TA
                                                          (treturn (MNonReact SM)
                                                              (tpair (tvar 2)
                                                                  (tabs 4 TI
                                                                      (tunfold
```

(*MReactT TI TO SM*) *TA TB*

(*tapp* (*tvar*

3) (*tvar* 4))

t2))))))))))),*Sto*).

Lemma *Congruence_STM_PauseBind* : ∀ *t1 t2 Sto TI TO SM T1 T2*,

value t1 →

value t2 →

(*tbind T2* (*tpause* (*MReactT TI TO SM*) *T1 t1*) *t2*,*Sto*) ~~~>*

(*tpause*

(*MReactT TI TO SM*)

T2

(*tbind*

(*TProd TO* (*TArrow TI* (*TMonadic* (*MReactT TI TO*

SM) *T2*)))

t1

(*tabs* 0 (*TProd TO* (*TArrow TI* (*TMonadic* (*MReactT*

TI TO SM) *T1*)))

(*tproj*

(*tvar* 0)

(*tabs* 1 *TO*

(*tabs* 2 (*TArrow TI* (*TMonadic* (*MReactT TI*

TO SM) *T1*))

(*treturn* (*MNonReact SM*)

(*tpair*

(*tvar* 1)

(*tabs* 3 *TI*

(*tbind*

T2

(*tapp* (*tvar* 2) (*tvar* 3))

t2))))))))),*Sto*).

Lemma *Congruence_AppAbs_Mo* : ∀ *x T t v Sto*,

value v →

((*tapp* (*tabs x T t*) *v*),*Sto*) ~~~>* ([*x*:=*v*]*t*,*Sto*).

Lemma *Congruence_Unfold1* : ∀ *TI TO SM TA TB t1 t1' t2*,

t1 ~~>* *t1'* →

tunfold (*MReactT TI TO SM*) *TA TB t1 t2* ~~>* *tunfold* (*MReactT TI TO SM*) *TA TB t1' t2*.

Lemma *Congruence_Unfold2* : ∀ *TI TO SM TA TB t1 t2 t2'*,

value t1 →

t2 ~~>* *t2'* →

tunfold (*MReactT TI TO SM*) *TA TB t1 t2* ~~>* *tunfold* (*MReactT TI TO SM*) *TA TB t1 t2'*.

Lemma *Congruence_STM_RunRe* : ∀ *T t1 t1' Sto Sto'*,

value t1 →

(*t1*,*Sto*) ~~~>* (*t1'*,*Sto'*) →

(*trunre T t1*,*Sto*) ~~~>* (*trunre T t1'*,*Sto'*).

Lemma *Congruence_STM_RunReRet* : ∀ *v TI TO SM TA Sto*,

value v →

(*trunre TA* (*treturn* (*MReactT TI TO SM*) *v*),*Sto*) ~~~>*

(*treturn* (*MNonReact SM*) (*tinl* (*TProd TO* (*TArrow TI* (*TMonadic* (*MReactT TI TO SM*) *TA*))) *v*),*Sto*).

Lemma *Congruence_STM_RunRePause* : ∀ *v TI TO SM TA Sto*,

value v →

(*trunre TA* (*tpause* (*MReactT TI TO SM*) *TA v*),*Sto*) ~~~>*

(*tbind* (*TSum TA* (*TProd TO* (*TArrow TI* (*TMonadic* (*MReactT TI TO SM*) *TA*))))

v

(*tabs* 0 (*TProd TO* (*TArrow TI* (*TMonadic* (*MReactT TI TO SM*) *TA*)))

(*treturn* (*MNonReact SM*) (*tinr TA* (*tvar* 0)))),*Sto*).

Injectivity Lemmas for some of the term constructors.

Theorem *tinl_stays_tinl* : ∀ *t T t'*,

tinl T t ~~>* *t'* →

∃ *t''*, *t'* = *tinl T t''*.

Theorem *tinr_stays_tinr* : ∀ *t T t'*,

tinr T t ~~>* *t'* →

∃ *t''*, *t'* = *tinr T t''*.

Lemma *treturn_step_inj* : ∀ *Mo t1 t2 Sto1 Sto2*,

(*treturn Mo t1*,*Sto1*) ~~~> (*treturn Mo t2*,*Sto2*) →

t1 ~~> *t2* ∧ *Sto1* = *Sto2*.

Lemma *treturn_step_inj_star* : ∀ *Mo t1 t2 Sto1 Sto2*,

(*treturn Mo t1*,*Sto1*) ~~~>* (*treturn Mo t2*,*Sto2*) →

t1 ~~>* *t2* ∧ *Sto1* = *Sto2*.

Lemma *step_return_no_change_store* : ∀ *t t' Mo Sto Sto'*,

(*treturn Mo t*,*Sto*) ~~~>* (*t'*,*Sto'*) → *Sto* = *Sto'*.

Theorem *pair_step_inj_l* : ∀ *t1 t2 v1 u2*,

value v1 →

```
                tpair t1 t2 ~~>* tpair v1 u2 →
                t1 ~~>* v1.

Theorem pair_step_inj_r : ∀ t1 t2 u1 v2,
                value v2 →
                tpair t1 t2 ~~>* tpair u1 v2 →
                t2 ~~>* v2.

Theorem tinl_step_inj : ∀ t T v,
                value v →
                tinl T t ~~>* tinl T v →
                t ~~>* v.

Theorem tinr_step_inj : ∀ t T v,
                value v →
                tinr T t ~~>* tinr T v →
                t ~~>* v.

Lemma step_value_eq : ∀ v v',
                value v →
                value v' →
                v ~~>* v' → v = v'.
```

B.8 Progress

```
Theorem progress' : ∀ t T,
                \empty |- t \in T →
                (value t ∨ ∃ t', t ~~> t') ∧
                (∀ M Tret,
                    T = TMonadic M Tret →
                    ∀ Sto,
```

store_matches_mo Sto M →

(*done_mo* (*t*,*Sto*) ∨ ∃ *t' Sto'*, ((*t*,*Sto*) ~~~> (*t'*,*Sto'*) ∧

same_length Sto Sto'))).

`Corollary` `progress` : ∀ *t T*,

empty |- *t* \`in` *T* →

(*value t* ∨ ∃ *t'*, *t* ~~> *t'*).

`Corollary` *progress_mo* : ∀ *co T*,

co |> *T* →

(*done_mo co* ∨ ∃ *co'*, *co* ~~~> *co'*).

`Corollary` *not_value_step* : ∀ *t T*,

empty |- *t* \`in` *T* →

¬ *value t* →

∃ *t'*, *t* ~~> *t'*.

`Corollary` *not_value_step_mo* : ∀ *t T Sto*,

empty |- *t* \`in` *T* →

¬ *value t* →

∃ *co'*, (*t*,*Sto*) ~~~> *co'*.

`Corollary` *not_done_step_mo* : ∀ *co T*,

co |> *T* →

¬ *done_mo co* →

∃ *co'*, *co* ~~~> *co'*.

B.9 Preservation

`Theorem` *preservation'* : ∀ *t T*,

empty |- *t* \`in` *T* →

((∀ *t'*, *t* ~~> *t'* → *empty* |- *t'* \`in` *T*)

$\wedge$ ($\forall$ *M Tret*,

T = TMonadic M Tret $\rightarrow$

$\forall$ *t' Sto Sto'*,

store_all_values Sto $\rightarrow$

store_matches_mo Sto M $\rightarrow$

(*t,Sto*) ~~~> (*t',Sto'*) $\rightarrow$

(*empty* |- *t'* \in *T* $\wedge$ *store_all_values Sto'* $\wedge$

store_matches_mo Sto' M))).

Corollary *preservation* : $\forall$ *T t*,

empty |- *t* \in *T* $\rightarrow$

$\forall$ *t'*, *t* ~~> *t'* $\rightarrow$ *empty* |- *t'* \in *T*.

Corollary *preservation_mo* : $\forall$ *T co*,

co |> *T* $\rightarrow$

$\forall$ *co'*,

co ~~~> *co'* $\rightarrow$ *co'* |> *T*.

Corollary *preservation_star* : $\forall$ *T t t'*,

t ~~>* *t'* $\rightarrow$

empty |- *t* \in *T* $\rightarrow$

empty |- *t'* \in *T*.

Corollary *preservation_mo_star* : $\forall$ *T t Sto t' Sto'*,

(*t,Sto*) ~~~>* (*t',Sto'*) $\rightarrow$

(*t,Sto*) |> *T* $\rightarrow$

(*t',Sto'*) |> *T*.

Corollary *preservation_mo_star_co* : $\forall$ *T co co'*,

co ~~~>* *co'* $\rightarrow$

co |> *T* $\rightarrow$

co' |> *T*.

B.10 Strong Normalization

Lambda calculus normal forms reduce no further.

`Definition` *normal_form* (*t*:*tm*) : `Prop` :=

¬ ∃ *t'*, *t* ~~> *t'*.

Because lambda calculus values do not single step reduce, it follows that if t is a value, then t is a normal form.

`Lemma` *value_norm_form* : ∀ (*t* : *tm*),

value t → *normal_form t*.

For any term t, *t* halts iff there exists a value t', such that t ~~>* t'.

`Definition` *halts* (*t*:*tm*) : `Prop` :=

∃ *t'*, *t* ~~>* *t'* ∧ *value t'*.

For any configuration c, *c* halts iff there exists a done configuration c', such that c ~~~>* c'.

`Definition` *halts_mo* (*co*:*configuration*) : `Prop` :=

∃ *co'*, *co* ~~~>* *co'* ∧ *done_mo co'*.

`Lemma` *values_halt* : ∀ *t*,

value t → *halts t*.

`Lemma` *done_halts* : ∀ *co*,

done_mo co → *halts_mo co*.

`CoInductive` *along_react* : (*store* → `Prop`) → (*tm* → `Prop`) → (*tm* → `Prop`) → (*tm* → `Prop`) → *configuration* → `Prop` :=

| *along_return* : ∀ (*PS*:*store* → `Prop`) (*PI PO PR*:*tm* → `Prop`) *TI TO SM t Sto t' Sto'*,

(*t*,*Sto*) ~~~>* (*treturn* (*MReactT TI TO SM*) *t'*,*Sto'*) →

```
                    value t' →
                    PR t' →
                    PS Sto' →
                    along_react PS PI PO PR (t,Sto)
  | along_pause : ∀ (PS:store → Prop) (PI PO PR:tm → Prop) TI TO SM T t Sto t' Sto' vl
vr Sto'',
                    (t,Sto) ~~>* (tpause (MReactT TI TO SM) T t',Sto') →
                    value t' →
                    value vl →
                    value vr →
                    PS Sto' →
                    PS Sto'' →
                    (t',Sto') ~~>* (treturn (MNonReact SM) (tpair vl vr),Sto'') →
                    PO vl →
                    (∀ t'', PI t'' → halts (tapp vr t'')) →
                    (∀ t'' Sto'',
                        PI t'' →
                        PS Sto'' →
                        along_react PS PI PO PR ((tapp vr t''),Sto'')) →
                    along_react PS PI PO PR (t,Sto).

Fixpoint R (T:Ty) (t:tm) {struct T} : Prop :=
  {} ⊢ t : T ∧ halts t ∧
  match T with
  | TArrow T1 T2 ⇒ ∀ s, R T1 s → R T2 (tapp t s)
  | TProd T1 T2 ⇒ ∃ t1 t2,
                    t ~~>* (tpair t1 t2) ∧
                    value t1 ∧
```

```
                                                    value t2 ∧
                                                    R T1 t1 ∧
                                                    R T2 t2
    | TSum T1 T2 ⇒ ∃ t',
                                                    value t' ∧
                                                    ((t ~~>* tinl T2 t' ∧ R T1 t') ∨
                                                     (t ~~>* tinr T1 t' ∧ R T2 t'))
    | TNil ⇒ True
    | TMonadic (MNonReact SM) T' ⇒ ∀ Sto,
                                                    Rsto SM Sto →
                                                    ∃ t' Sto',
                                                      (t,Sto) ~~~>* (treturn (MNonReact SM)
t',Sto') ∧
                                                      value t' ∧
                                                      R T' t' ∧
                                                      Rsto SM Sto'
    | TMonadic (MReactT TI TO SM) T' ⇒ ∀ Sto,
                                                    Rsto SM Sto →
                                                    ∃ t' Sto',
                                                      (t,Sto) ~~~>* (t',Sto') ∧
                                                      value t' ∧
                                                      Rsto SM Sto' ∧
                                                       along_react (Rsto SM) (R TI) (R TO)
(R T') (t',Sto')
      end
with Rsto (SM:SMo) (Sto:store) {struct SM} : Prop :=
    store_all_values Sto ∧
```

```
    store_matches_mo Sto (MNonReact SM) ∧
    match SM with
    | MIdentity ⇒ True
    | MStateT T b SM' ⇒ ∃ t Sto',
                                    R T t ∧
                                    Rsto SM' Sto' ∧
                                    Sto = t::Sto'
    end.

Lemma Rsto_all_values : ∀ {SM} {Sto},
      Rsto SM Sto → store_all_values Sto.

Lemma R_halts : ∀ {T} {t},
      R T t → halts t.

Lemma R_halts_nonreact : ∀ T SM t Sto,
      R (TMonadic (MNonReact SM) T) t →
            Rsto SM Sto →
        halts_mo (t, Sto).

Lemma R_halts_react : ∀ TI TO SM T t Sto,
      R (TMonadic (MReactT TI TO SM) T) t →
      Rsto SM Sto →
      halts_mo (t,Sto).

Lemma R_typable_empty : ∀ {T} {t},
                  R T t →
      {} ⊢ t : T.
```

Facts conerning Rsto and the permissiveness ordering on state monads.

```
Lemma Rsto_matches : ∀ Sto SM,
```

Rsto SM Sto →

store_matches_mo Sto (*MNonReact SM*).

Lemma *Rsto_less_permissive* : ∀ *SM SM' Sto*,

smo_less_permissive SM SM' →

Rsto SM Sto →

Rsto SM' Sto.

Lemma *Rsto_more_permissive* : ∀ *SM SM' Sto*,

smo_less_permissive SM' SM →

Rsto SM Sto →

Rsto SM' Sto.

Lemma *step_preserves_halting* : ∀ *t t'*,

(*t* ~~> *t'*) → (*halts t* ↔ *halts t'*).

Lemma *multistep_preserves_halting* : ∀ *t t'*,

(*t* ~~>* *t'*) → (*halts t* ↔ *halts t'*).

Lemma *step_preserves_along* :

∀ *t Sto t' Sto'* (*PS*:*store* → Prop) *PI PO PR*,

(*t*,*Sto*) ~~~> (*t'*,*Sto'*) →

along_react PS PI PO PR (*t'*,*Sto'*) →

along_react PS PI PO PR (*t*, *Sto*).

Lemma *step_preserves_along'* :

∀ *t Sto t' Sto' TI TO TP M*,

(*t*,*Sto*) ~~~> (*t'*,*Sto'*) →

along_react (*Rsto M*) (*R TI*) (*R TO*) (*R TP*) (*t*,*Sto*) →

along_react (*Rsto M*) (*R TI*) (*R TO*) (*R TP*) (*t'*,*Sto'*).

Lemma *step_preserves_along_star* :

∀ *t Sto t' Sto'* (*PS*:*store* → Prop) *PI PO PR*,

(*t*,*Sto*) ~~~>* (*t'*,*Sto'*) →

along_react PS PI PO PR (*t'*,*Sto'*) →

along_react PS PI PO PR (*t*, *Sto*).

Lemma *step_preserves_along_star'* :

∀ *t Sto t' Sto' TI TO TP M*,

(*t*,*Sto*) ~~~>* (*t'*,*Sto'*) →

along_react (*Rsto M*) (*R TI*) (*R TO*) (*R TP*) (*t*, *Sto*) →

along_react (*Rsto M*) (*R TI*) (*R TO*) (*R TP*) (*t'*,*Sto'*).

Lemma *step_along_react* :

∀ *t Sto t' Sto' TI TO TP M*,

(*t*,*Sto*) ~~~> (*t'*,*Sto'*) →

(*along_react* (*Rsto M*) (*R TI*) (*R TO*) (*R TP*) (*t*, *Sto*) ↔

along_react (*Rsto M*) (*R TI*) (*R TO*) (*R TP*) (*t'*,*Sto'*)).

Lemma *step_along_react_star* :

∀ *t Sto t' Sto' TI TO TP M*,

(*t*,*Sto*) ~~~>* (*t'*,*Sto'*) →

(*along_react* (*Rsto M*) (*R TI*) (*R TO*) (*R TP*) (*t*, *Sto*) ↔

along_react (*Rsto M*) (*R TI*) (*R TO*) (*R TP*) (*t'*,*Sto'*)).

Lemma *step_preserves_R* : ∀ *T t t'*,

(*t* ~~> *t'*) →

R T t →

R T t'.

Lemma *multistep_preserves_R* : ∀ *T t t'*,

(*t* ~~>* *t'*) →

R T t →

R T t'.

```
Lemma step_preserves_R' : ∀ T t t',
      {} ⊢ t : T →
                  t ~~> t' →
                  R T t' →
                  R T t.

Lemma multistep_preserves_R' : ∀ T t t',
      {} ⊢ t : T →
            (t ~~>* t') →
                  R T t' →
                  R T t.

Definition env := list (id × tm).

Fixpoint closed_env (env:env) {struct env} :=
          match env with
          | nil ⇒ True
          | (x,t)::env' ⇒ closed t ∧ closed_env env'
          end.

Fixpoint msubst (ss:env) (t:tm) {struct ss} : tm :=
          match ss with
          | nil ⇒ t
          | ((x,s)::ss') ⇒ msubst ss' ([x:=s]t)
          end.

Definition tass := list (id × Ty).

Fixpoint mextend (Gamma : context) (xts : tass) :=
          match xts with
          | nil ⇒ Gamma
          | ((x,v)::xts') ⇒ extend (mextend Gamma xts') x v
```

```
        end.

Fixpoint lookup {X:Set} (k : id) (l : list (id × X)) {struct l} : option X :=
        match l with
        | nil ⇒ None
        | (j,x) :: l' ⇒ if eq_id_dec j k then Some x else lookup k l'
        end.

Fixpoint drop {X:Set} (n:id) (nxs:list (id × X)) {struct nxs} : list (id × X) :=
        match nxs with
        | nil ⇒ nil
        | ((n',x)::nxs') ⇒ if eq_id_dec n' n then drop n nxs' else (n',x)::(drop n nxs')
        end.

Inductive instantiation : tass → env → Prop :=
  | V_nil : instantiation nil nil
  | V_cons : ∀ x T v c e,
                                value v →
                                  R T v →
            instantiation c e →
            instantiation ((x,T)::c) ((x,v)::e).

Lemma mextend_lookup : ∀ (c:tass) (x:id),
        lookup x c = (mextend empty c) x.

Lemma mextend_drop : ∀ (c:tass) Gamma x x',
        mextend Gamma (drop x c) x' = if eq_id_dec x x' then Gamma x' else mextend
Gamma c x'.

Lemma instantiation_domains_match: ∀ {c} {e},
        instantiation c e →
        ∀ {x} {T}, lookup x c = Some T → ∃ t, lookup x e = Some t.
```

```
Lemma instantiation_env_closed : ∀ c e,
      instantiation c e → closed_env e.

Lemma instantiation_R : ∀ c e,
      instantiation c e →
                              ∀ x t T,
      lookup x c = Some T →
      lookup x e = Some t →
                        R T t.

Lemma instantiation_drop : ∀ c env,
   instantiation c env → ∀ x, instantiation (drop x c) (drop x env).

Lemma mextend_empty_lookup : ∀ c x, (mextend empty c) x = lookup x c.

Lemma msubst_closed: ∀ t,
                  closed t →
                        ∀ ss,
                           msubst ss t = t.

Lemma msubst_preserves_typing : ∀ c e,
      instantiation c e →
      ∀ Gamma t S, (mextend Gamma c) ⊢ t : S →
      Gamma ⊢ (msubst e t) : S.

Lemma subst_msubst: ∀ env x v t,
               closed v →
      closed_env env →
      msubst env ([x:=v]t) = [x:=v](msubst (drop x env) t).

Lemma msubst_var: ∀ ss x, closed_env ss →
      msubst ss (tvar x) =
      match lookup x ss with
```

| *Some t* ⇒ *t*

| *None* ⇒ *tvar x*

end.

Lemma *msubst_abs*: ∀ *ss x T t*,

msubst ss (*tabs x T t*) = *tabs x T* (*msubst* (*drop x ss*) *t*).

Lemma *msubst_app* : ∀ *ss t1 t2*,

msubst ss (*tapp t1 t2*) = *tapp* (*msubst ss t1*) (*msubst ss t2*).

Lemma *msubst_pair* : ∀ *ss t1 t2*,

msubst ss (*tpair t1 t2*) = *tpair* (*msubst ss t1*) (*msubst ss t2*).

Lemma *msubst_proj* : ∀ *ss t1 t2*,

msubst ss (*tproj t1 t2*) = *tproj* (*msubst ss t1*) (*msubst ss t2*).

Lemma *msubst_tinl* : ∀ *ss T t*,

msubst ss (*tinl T t*) = *tinl T* (*msubst ss t*).

Lemma *msubst_tinr* : ∀ *ss T t*,

msubst ss (*tinr T t*) = *tinr T* (*msubst ss t*).

Lemma *msubst_tcase* : ∀ *ss t1 t2 t3*,

msubst ss (*tcase t1 t2 t3*) = *tcase* (*msubst ss t1*) (*msubst ss t2*) (*msubst ss t3*).

Lemma *msubst_return* : ∀ *ss M t*,

msubst ss (*treturn M t*) = *treturn M* (*msubst ss t*).

Lemma *msubst_bind* : ∀ *ss T t1 t2*,

msubst ss (*tbind T t1 t2*) = *tbind T* (*msubst ss t1*) (*msubst ss t2*).

Lemma *msubst_lift* : ∀ *ss M t*,

msubst ss (*tlift M t*) = *tlift M* (*msubst ss t*).

Lemma *msubst_elevate* : ∀ *ss M t*,

msubst ss (*televate M t*) = *televate M* (*msubst ss t*).

```
Lemma msubst_tunit : ∀ ss,
      msubst ss tunit = tunit.

Lemma msubst_get : ∀ ss M,
      msubst ss (tget M) = tget M.

Lemma msubst_tput : ∀ ss t M,
      msubst ss (tput M t) = tput M (msubst ss t).

Lemma msubst_trunst : ∀ ss t1 t2,
                              msubst ss (trunst t1 t2) = trunst (msubst ss t1) (msubst ss t2).

Lemma msubst_trunid : ∀ ss t,
                              msubst ss (trunid t) = trunid (msubst ss t).

Lemma msubst_tpause : ∀ ss t M T,
                               msubst ss (tpause M T t) = tpause M T (msubst ss t).

Lemma msubst_tunfold : ∀ ss t1 t2 TA TB M,
                               msubst ss (tunfold M TA TB t1 t2) = tunfold M TA TB
(msubst ss t1) (msubst ss t2).

Lemma msubst_trunre : ∀ ss t T,
                              msubst ss (trunre T t) = trunre T (msubst ss t).

Lemma msubst_rewrite : ∀ ss t,
      msubst ss t = (match t with
                           | (tabs x T t') ⇒ tabs x T (msubst (drop x ss) t')
                           | (tapp t1 t2) ⇒ tapp (msubst ss t1) (msubst ss t2)
                           | (tpair t1 t2) ⇒ tpair (msubst ss t1) (msubst ss t2)
                           | (tproj t1 t2) ⇒ tproj (msubst ss t1) (msubst ss t2)
                           | (tinl T t') ⇒ tinl T (msubst ss t')
                           | (tinr T t') ⇒ tinr T (msubst ss t')
                           | (tcase t1 t2 t3) ⇒ tcase (msubst ss t1) (msubst ss t2) (msubst ss
```

t3)

| (*treturn M t'*) ⇒ *treturn M* (*msubst ss t'*)

| (*tbind T t1 t2*) ⇒ *tbind T* (*msubst ss t1*) (*msubst ss t2*)

| (*tlift M t'*) ⇒ *tlift M* (*msubst ss t'*)

| (*televate M t'*) ⇒ *televate M* (*msubst ss t'*)

| *tunit* ⇒ *tunit*

| (*tget M*) ⇒ *tget M*

| (*tput M t'*) ⇒ *tput M* (*msubst ss t'*)

| (*trunst t1 t2*) ⇒ *trunst* (*msubst ss t1*) (*msubst ss t2*)

| (*trunid t'*) ⇒ *trunid* (*msubst ss t'*)

| (*tpause M T t'*) ⇒ *tpause M T* (*msubst ss t'*)

| (*tunfold M TA TB t1 t2*) ⇒ *tunfold M TA TB* (*msubst ss t1*)
(*msubst ss t2*)

| (*trunre T t'*) ⇒ *trunre T* (*msubst ss t'*)

| _ ⇒ *msubst ss t*

end).

Lemma *step_mo_still_values_star* : ∀ *t t' Sto Sto'*,

store_all_values Sto →

(*t,Sto*) ~~~>* (*t',Sto'*) →

store_all_values Sto'.

Lemma *tbind_along* : ∀ *t Sto SM TI TO T1 T2 t1 t2 Sto' Sto''*,

{} ⊢ *t* : *TMonadic* (*MReactT TI TO SM*) *T2* →

store_all_values Sto →

store_matches_mo Sto (*MReactT TI TO SM*) →

(*t,Sto*) ~~~>* (*tbind T2 t1 t2,Sto'*) →

{} ⊢ *t1* : *TMonadic* (*MReactT TI TO SM*) *T1* →

store_all_values Sto' →

store_matches_mo Sto' (MReactT TI TO SM) →

{} ⊢ t2 : TArrow T1 (TMonadic (MReactT TI TO SM) T2) →

value t1 →

value t2 →

Rsto SM Sto'' →

along_react (Rsto SM) (R TI) (R TO) (R T1) (t1,Sto') →

(∀ t' Sto'',

R T1 t' →

Rsto SM Sto'' →

along_react (Rsto SM) (R TI) (R TO) (R T2) ((tapp t2 t'), Sto'')) →

along_react (Rsto SM) (R TI) (R TO) (R T2) (t,Sto).

Lemma *tunfold_along : ∀ t Sto TA TB TI TO SM t1 t2 Sto',*

((t,Sto) ~~~> (tunfold (MReactT TI TO SM) TA TB t1 t2,Sto')) →*

R TB t1 →

R (TArrow TB (TMonadic (MNonReact SM) (TSum TA (TProd TO (TArrow TI TB))))) t2 →

Rsto SM Sto' →

along_react (Rsto SM) (R TI) (R TO) (R TA) (t,Sto).

Lemma *tbind_R : ∀ M T1 T2 t t',*

R (TMonadic M T1) t →

R (TArrow T1 (TMonadic M T2)) t' →

R (TMonadic M T2) (tbind T2 t t').

Lemma *msubst_R : ∀ c env t T,*

(mextend empty c) ⊢ t : T →

instantiation c env →

```
    R T (msubst env t).

Theorem normalization : ∀ (t:tm) (T:Ty),
      {} ⊢ t : T → halts t.

Lemma WT_Configs_RSto_react : ∀ (t:tm) (Sto:store) (TI TO T:Ty) (SM:SMo),
                              (t, Sto) |> TMonadic (MReactT TI TO SM) T →
                              Rsto SM Sto.

Lemma WT_Configs_RSto_nonreact : ∀ (t:tm) (Sto:store) (T : Ty) (SM:SMo),
                                 (t, Sto) |> TMonadic (MNonReact SM) T →
                                 Rsto SM Sto.

Theorem normalization_mo : ∀ (t:tm) (Sto:store) (T:Ty) (M:Mo),
      (t, Sto) |> TMonadic M T →
                        halts_mo (t, Sto).

Theorem non_reactive_done_mo : ∀ t Sto T SM,
                               (t,Sto) |> TMonadic (MNonReact SM) T →
                               ∀ v Sto',
                                  done_mo (v,Sto') →
                                  (t,Sto) ~~~>* (v,Sto') →
                                  ∃ v',
                                  v = treturn (MNonReact SM) v'.
```

B.11 Effects

```
Inductive same_where_no_write : Mo → store → store → Prop :=
  | SWNW_Identity : same_where_no_write (MNonReact MIdentity) nil nil
  | SWNW_ReactT :
        ∀ Sto Sto' SM T T',
           same_where_no_write (MNonReact SM) Sto Sto' →
```

```
      same_where_no_write (MReactT T T' SM) Sto Sto'
  | SWNW_StateT_None :
      ∀ Sto Sto' SM s T,
        same_where_no_write (MNonReact SM) Sto Sto' →
        same_where_no_write (MNonReact (MStateT T EffNone SM)) (s::Sto) (s::Sto')
  | SWNW_StateT_R :
      ∀ Sto Sto' SM s T,
        same_where_no_write (MNonReact SM) Sto Sto' →
        same_where_no_write (MNonReact (MStateT T EffR SM)) (s::Sto) (s::Sto')
  | SWNW_StateT_W :
      ∀ Sto Sto' SM s s' T,
        same_where_no_write (MNonReact SM) Sto Sto' →
        same_where_no_write (MNonReact (MStateT T EffW SM)) (s::Sto) (s'::Sto')
  | SWNW_StateT_RW :
      ∀ Sto Sto' SM s s' T,
        same_where_no_write (MNonReact SM) Sto Sto' →
        same_where_no_write (MNonReact (MStateT T EffRW SM)) (s::Sto) (s'::Sto').

Lemma same_where_no_write_refl : ∀ M Sto,
                                   store_matches_mo Sto M →
                                   same_where_no_write M Sto Sto
with same_where_no_write_refl_sm : ∀ SM Sto,
                                     store_matches_mo Sto (MNonReact SM) →
                                     same_where_no_write (MNonReact SM)
Sto Sto.

Lemma same_where_no_write_trans : ∀ M Sto1 Sto2 Sto3,
                                    same_where_no_write M Sto1 Sto2 →
                                    same_where_no_write M Sto2 Sto3 →
```

same_where_no_write M Sto1 Sto3

`with` *same_where_no_write_trans_sm* : ∀ *SM Sto1 Sto2 Sto3*,

same_where_no_write (*MNonReact SM*)
Sto1 Sto2 →

same_where_no_write (*MNonReact SM*)
Sto2 Sto3 →

same_where_no_write (*MNonReact SM*)
Sto1 Sto3.

`Lemma` *same_where_no_write_less_permissive* :

∀ *SM1 SM2 Sto Sto'*,

smo_less_permissive SM1 SM2 →

same_where_no_write (*MNonReact SM1*) *Sto Sto'* →

same_where_no_write (*MNonReact SM2*) *Sto Sto'*.

`Theorem` *no_forbidden_updates* :

∀ *t Sto M T*,

(*t*,*Sto*) |> *TMonadic M T* →

∀ *t' Sto'*,

(*t*,*Sto*) ~~> (*t'*,*Sto'*) →

same_where_no_write M Sto Sto'.

`Theorem` *no_forbidden_updates_star* :

∀ *t Sto M T*,

(*t*,*Sto*) |> *TMonadic M T* →

∀ *t' Sto'*,

(*t*,*Sto*) ~~>* (*t'*,*Sto'*) →

same_where_no_write M Sto Sto'.

`Inductive` *same_where_read* : *Mo* → *store* → *store* → `Prop` :=

| *SWR_Identity* : *same_where_read* (*MNonReact MIdentity*) *nil nil*

```
| SWR_ReactT :
    ∀ Sto Sto' SM T T',
      same_where_read (MNonReact SM) Sto Sto' →
      same_where_read (MReactT T T' SM) Sto Sto'
| SWR_StateT_None :
    ∀ Sto Sto' SM s s' T,
      same_where_read (MNonReact SM) Sto Sto' →
      same_where_read (MNonReact (MStateT T EffNone SM)) (s::Sto) (s'::Sto')
| SWR_StateT_R :
    ∀ Sto Sto' SM s T,
      same_where_read (MNonReact SM) Sto Sto' →
      same_where_read (MNonReact (MStateT T EffR SM)) (s::Sto) (s::Sto')
| SWR_StateT_W :
    ∀ Sto Sto' SM s s' T,
      same_where_read (MNonReact SM) Sto Sto' →
      same_where_read (MNonReact (MStateT T EffW SM)) (s::Sto) (s'::Sto')
| SWR_StateT_RW :
    ∀ Sto Sto' SM s T,
      same_where_read (MNonReact SM) Sto Sto' →
      same_where_read (MNonReact (MStateT T EffRW SM)) (s::Sto) (s::Sto').

Inductive write_consistent : (store×store) → (store×store) → Prop :=
| WC_Identity : write_consistent (nil,nil) (nil,nil)
| WC_Unchanged : ∀ Sto1 Sto2 Sto1' Sto2' s1 s2,
                   write_consistent (Sto1,Sto2) (Sto1',Sto2') →
                   write_consistent (s1::Sto1,s2::Sto2) (s1::Sto1',s2::Sto2')
| WC_Changed : ∀ Sto1 Sto2 Sto1' Sto2' s1 s2 s,
                   write_consistent (Sto1,Sto2) (Sto1',Sto2') →
```

write_consistent (*s1*::*Sto1*,*s2*::*Sto2*) (*s*::*Sto1'*,*s*::*Sto2'*).

```
Lemma write_consistent_same_length_refl :
  ∀ Sto1 Sto2,
    same_length Sto1 Sto2 →
    write_consistent (Sto1,Sto2) (Sto1,Sto2).

Lemma same_monad_same_length :
  ∀ M Sto1 Sto2,
    store_matches_mo Sto1 M →
    store_matches_mo Sto2 M →
    same_length Sto1 Sto2
with same_monad_same_length_sm :
  ∀ SM Sto1 Sto2,
    store_matches_mo Sto1 (MNonReact SM) →
    store_matches_mo Sto2 (MNonReact SM) →
    same_length Sto1 Sto2.

Lemma same_where_read_same_length :
  ∀ M Sto1 Sto2,
    same_where_read M Sto1 Sto2 →
    same_length Sto1 Sto2
with same_where_read_same_length_sm :
  ∀ SM Sto1 Sto2,
    same_where_read (MNonReact SM) Sto1 Sto2 →
    same_length Sto1 Sto2.

Lemma same_where_read_tail :
  ∀ T E SM s1 Sto1 s2 Sto2,
    same_where_read (MNonReact (MStateT T E SM)) (s1::Sto1) (s2::Sto2) →
```

```
    same_where_read (MNonReact SM) Sto1 Sto2.

Lemma same_where_read_head :
  ∀ T E SM s1 Sto1 s2 Sto2,
    Eff_lt EffR E →
    same_where_read (MNonReact (MStateT T E SM)) (s1::Sto1) (s2::Sto2) →
    s1 = s2.

Lemma same_where_read_less_permissive :
  ∀ SM1 SM2,
    smo_less_permissive SM1 SM2 →
  ∀ Sto1 Sto2,
    same_where_read (MNonReact SM2) Sto1 Sto2 →
    same_where_read (MNonReact SM1) Sto1 Sto2.

Theorem no_forbidden_reads : ∀ t Sto1 Sto2 T M,
                                (t,Sto1) |> TMonadic M T →
                                (t,Sto2) |> TMonadic M T →
                                same_where_read M Sto1 Sto2 →
                              ∀ t1' t2' Sto1' Sto2',
                                (t,Sto1) ~~~> (t1',Sto1') →
                                (t,Sto2) ~~~> (t2',Sto2') →
                                t1' = t2' ∧ write_consistent (Sto1,Sto2) (Sto1',Sto2').
```

Read Only Predicate:

```
Inductive read_only : SMo → Prop :=
 | Ronly_ST : ∀ T E SM,
                Eff_lt E EffR →
                read_only SM →
                read_only (MStateT T E SM)
```

```
| Ronly_Id : read_only MIdentity.

Theorem read_only_eff' : ∀ T E SM,
                         read_only (MStateT T E SM) →
                             Eff_lt E EffR.

Theorem read_only_smo : ∀ T E SM,
                        read_only (MStateT T E SM) →
                            read_only SM.

Theorem read_only_smo_lp' : ∀ SM1 SM2,
                            smo_less_permissive SM1 SM2 →
                            read_only SM2 →
                            read_only SM1.

Theorem read_only_smo_lp_ST : ∀ T E SM1 SM2,
                              smo_less_permissive SM1 SM2 →
                              read_only (MStateT T E SM2) →
                              read_only (MStateT T E SM1).

Theorem read_only_elim_w : ∀ T SM, read_only (MStateT T EffW SM) → False.

Theorem read_only_elim_rw : ∀ T SM, read_only (MStateT T EffRW SM) → False.

Theorem read_only_elim_base:
  ∀ SM T E,
    ¬ read_only SM →
        ¬ read_only (MStateT T E SM).

Theorem read_only_st_dec : ∀ T E SM,
        read_only (MStateT T E SM) ∨ ¬ read_only (MStateT T E SM).

Theorem read_only_dec : ∀ SM, read_only SM ∨ ¬ read_only SM.

Lemma same_where_no_write_read_only_eq :
```

∀ *SM*,

read_only SM →

∀ *Sto Sto'*,

same_where_no_write (*MNonReact SM*) *Sto Sto'* → *Sto* = *Sto'*.

`Theorem` *step_read_only_no_change* : ∀ *SM*,

read_only SM →

∀ *t Sto t' Sto' T*,

(*t*,*Sto*) ~~~> (*t'*,*Sto'*) →

(*t*,*Sto*) |> *TMonadic* (*MNonReact SM*) *T* →

Sto = *Sto'*.

`Theorem` *step_read_only_no_change_star* : ∀ *SM*,

read_only SM →

∀ *t Sto t' Sto' T*,

(*t*,*Sto*) ~~~>* (*t'*,*Sto'*) →

(*t*,*Sto*) |> *TMonadic* (*MNonReact SM*) *T* →

Sto = *Sto'*.

B.12 Monad Laws

`Theorem` *left_unit* : ∀ (*M* : *Mo*) (*T1 T2* : *Ty*) (*t1 t2* : *tm*),

empty |– *t1* \`in` *T1* →

empty |– *t2* \`in` *TArrow T1* (*TMonadic M T2*) →

(*tapp t2 t1*) =[*TMonadic M T2*]= (*treturn M t1* »=[*T2*] *t2*).

`Lemma` *right_unit* : ∀ (*t* : *tm*) (*SM* : *SMo*) (*T* : *Ty*) (*x* : *id*),

empty |– *t* \`in` *TMonadic* (*MNonReact SM*) *T* →

t =[(*TMonadic* (*MNonReact SM*) *T*)]= (*t* »=[*T*] \(*x*:*T*)(*treturn* (*MNonReact SM*) (*tvar x*))).

Lemma *associativity_of_bind* : ∀ (*SM* : *SMo*) (*T1 T2 T3* : *Ty*) (*f g h* : *tm*) (*x* : *id*),

empty |– *f* \in *TMonadic* (*MNonReact SM*) *T1* →

empty |– *g* \in *TArrow T1* (*TMonadic* (*MNonReact SM*) *T2*) →

empty |– *h* \in *TArrow T2* (*TMonadic* (*MNonReact SM*) *T3*) →

((*f* »=[*T2*] *g*) »=[*T3*] *h*) =[*TMonadic* (*MNonReact SM*) *T3*]= (*f* »=[*T3*] (\(*x*:*T1*) (*tapp g* (*tvar x*) »=[*T3*] *h*))).

Reactive Monad Axioms

Axiom *RE_right_unit* : ∀ (*t* : *tm*) (*SM* : *SMo*) (*TI TO T* : *Ty*) (*x* : *id*),

empty |– *t* \in *TMonadic* (*MReactT TI TO SM*) *T* →

t =[(*TMonadic* (*MReactT TI TO SM*) *T*)]= (*t* »=[*T*] \(*x*:*T*)(*treturn* (*MReactT TI TO SM*) (*tvar x*))).

Axiom *RE_associativity_of_bind* : ∀ (*SM* : *SMo*) (*TI TO T1 T2 T3* : *Ty*) (*f g h* : *tm*) (*x* : *id*),

empty |– *f* \in *TMonadic* (*MReactT TI TO SM*) *T1* →

empty |– *g* \in *TArrow T1* (*TMonadic* (*MReactT TI TO SM*) *T2*) →

empty |– *h* \in *TArrow T2* (*TMonadic* (*MReactT TI TO SM*) *T3*) →

((*f* »=[*T2*] *g*) »=[*T3*] *h*) =[*TMonadic* (*MReactT TI TO SM*) *T3*]= (*f* »=[*T3*] (\(*x*:*T1*) (*tapp g* (*tvar x*) »=[*T3*] *h*))).

B.12.1 Monad Transformer Laws

Lemma *lift_return_nonreact* : ∀ *t T TS E SM*,

empty |– *t* \in *T* →

(*tlift* (*MNonReact* (*MStateT TS E SM*)) (*treturn* (*MNonReact SM*) *t*))

=[*TMonadic* (*MNonReact* (*MStateT TS E SM*)) *T*]=

(*treturn* (*MNonReact* (*MStateT TS E SM*)) *t*).

Lemma *lift_return_react* : ∀ *t SM T TI TO*,

empty |– *t* \in *T* →

(*tlift* (*MReactT TI TO SM*) (*treturn* (*MNonReact SM*) *t*)) =[*TMonadic* (*MReactT TI TO SM*) *T*]= (*treturn* (*MReactT TI TO SM*) *t*).

Lemma *lift_bind_nonreact* : ∀ *x t1 t2 T1 T2 TS E SM*,

empty |– *t1* \in *TMonadic* (*MNonReact SM*) *T1* →

empty |– *t2* \in *TArrow T1* (*TMonadic* (*MNonReact SM*) *T2*) →

(*tlift* (*MNonReact* (*MStateT TS E SM*)) (*t1* »=[*T2*] *t2*))

=[*TMonadic* (*MNonReact* (*MStateT TS E SM*)) *T2*]=

((*tlift* (*MNonReact* (*MStateT TS E SM*)) *t1*) »=[*T2*] \(*x*:*T1*) (*tlift* (*MNonReact* (*MStateT TS E SM*)) (*tapp t2* (*tvar x*)))).

Lemma *lift_bind_react* : ∀ *x t1 t2 T1 T2 TI TO SM*,

empty |– *t1* \in *TMonadic* (*MNonReact SM*) *T1* →

empty |– *t2* \in *TArrow T1* (*TMonadic* (*MNonReact SM*) *T2*) →

(*tlift* (*MReactT TI TO SM*) (*t1* »=[*T2*] *t2*))

=[*TMonadic* (*MReactT TI TO SM*) *T2*]=

((*tlift* (*MReactT TI TO SM*) *t1*) »=[*T2*] \(*x*:*T1*) (*tlift* (*MReactT TI TO SM*) (*tapp t2* (*tvar x*)))).

B.12.2 Null Bind

Notation "t1 »[T1 , T2] t2" := (*t1* »=[*T2*] \(6:*T1*) *t2*)

(at level 40, *T* at level 99, *T'* at level 99, *format* "'[hv ' t1 »[T1 , T2] t2 ']'").

Theorem *generic_null_bind* : ∀ *t t' M T T'*,

empty |– *t* \in *TMonadic M T* →

empty |– *t'* \in *TMonadic M T'* →

∀ *b* : *id*,

(*t* »=[*T'*] (\(*b*:*T*) *t'*)) =[*TMonadic M T'*]= (*t* »=[*T*,*T'*] *t'*).

B.12.3 Stateful Computations

Theorem *ST_put_put* : ∀ *s s' E SM T*,

Eff_lt EffW E →

empty |– *s* \in *T* →

empty |– *s'* \in *T* →

((*tput* (*MStateT T E SM*) *s*) »[*TNil*,*TNil*] (*tput* (*MStateT T E SM*) *s'*))

=[*TMonadic* (*MNonReact* (*MStateT T E SM*)) *TNil*]= *tput* (*MStateT T E SM*) *s'*.

Theorem *ST_put_get* : ∀ *s T SM*,

empty |– *s* \in *T* →

(*tput* (*MStateT T EffRW SM*) *s*) »[*TNil*,*T*] (*tget* (*MStateT T EffRW SM*))

=[*TMonadic* (*MNonReact* (*MStateT T EffRW SM*)) *T*]=

(*tput* (*MStateT T EffRW SM*) *s*) »[*TNil*,*T*] (*treturn* (*MNonReact* (*MStateT T EffRW SM*)) *s*).

Theorem *ST_get_get* : ∀ *T E SM x y z*,

Eff_lt EffR E →

tget (*MStateT T E SM*) »=[*TProd T T*] (\(*x*:*T*) (*tget* (*MStateT T E SM*) »=[*TProd T T*] (\(*y*:*T*) (*treturn* (*MNonReact* (*MStateT T E SM*)) (*tpair* (*tvar x*) (*tvar y*))))))

=[*TMonadic* (*MNonReact* (*MStateT T E SM*)) (*TProd T T*)]=

tget (*MStateT T E SM*) »=[*TProd T T*] (\(*z*:*T*) (*treturn* (*MNonReact* (*MStateT T E SM*)) (*tpair* (*tvar z*) (*tvar z*)))).

Theorem *elevate_absorb* : ∀ *t t' SM SM' T T'*,

smo_less_permissive SM SM' →

empty |– *t* \in *TMonadic* (*MNonReact SM*) *T* →

```
\empty |- t' \in TMonadic (MNonReact SM') T' →
                                              read_only SM' →
((televate SM' t »[T,T'] t') =[TMonadic (MNonReact SM') T']= t').
```

VITA

Thomas Reynolds was born in Baton Rouge, Louisiana on October 16^{th}, 1982. He received his B.S. in Political Science from Illinois State University in May of 2007, an M.A. in Philosophy from Texas Tech University in May of 2010 and an M.A. in Philosophy from the University of Missouri in December of 2013. He completed his PhD in Computer Science at the University of Missouri in December of 2019.

In September of 2011, Thomas married Jenna Marie Quick. Their daughter, Elliette Quinn Reynolds, was born July 15^{th}, 2017. They are expecting a second child, Everett Boone Reynolds, to be born in January of 2020. Thomas and his family currently reside in Mount Vernon, Illinois.