# Indexed Reactive Programming

Stefan Knudsen

Master of Science

School of Computer Science

McGill University

Montreal,Quebec

2019-07-21

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Science

# ACKNOWLEDGEMENTS

I would like to thank my family for their support throughout this period of my life. Thank you to all of the students who have come through the lab, and to my fellow master's students Shawn Otis and Rohan Jacob Rao. Thank you especially to Francisco Ferreira, David Thibodeau, and Andrew Cave, for the many white-board explanations and support. I would also like to extend a thank you to my TAs and friends in the compiler lab, Erick Lavoie, Fez Khan, Hanfeng Chen, Prabhjot Sandhu, Steven Thephsourinthone, and Vincent Foley. Finally, I am incredibly grateful to Professor Pientka for taking me on as a student, for her patience, and for what I have learned.

◊ ◊ ◊

# ABSTRACT

Linear Temporal Logic (LTL) is a logic for reasoning about time. It has been used effectively to formally verify systems since it can prove the absence of certain bugs. Under the Curry-Howard correspondence, LTL corresponds to functional reactive programming, a paradigm that handles dataflow propogation. Indexed types, whose Curry-Howard analog is first-order logic, allow types to depend on terms from some index language, and can also ensure program behaviour by virtue of compilation. This thesis introduces Jackrabbit, a reactive programming language with indexed types. We explain the syntax and type system, present a few motivating examples, and prove type safety.

# RÉSUMÉ

La logique temporelle linéaire (LTL) est une logique pour raisonné à propos du temps. Elle est utile pour vérifier les systèmes parce qu'elle est capabale de prouver l'absence de certaines bugs. Sous la correspondence Curry-Howard, l'LTL correspond à la programmation réactive fonctionnelle, une paradigme qui traite la propagation des stream de données. Sous cette même correspondence, les types indexes correpondent à la logique de première ordre, qui permet les types de dépendre sur des termes d'une langage indexe. Les indexes permettent aussi de garantir des propriétés dès compilation. Cette thèse introduit Jackrabbit, une langage réactive avec des types indexes. On explique le syntax et la système de typage, on présente quelques exemples motivants, et on donne une preuve de la sûreté du typage.

## TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1
# Introduction

## 1.1 Constraint

A scientific theory is a model. When modeling the world, it can err in two ways. The first is that it can be too strict, not being able to make claims that accurately model the world and that therefore falsify the theory. For example, the theory that planets travel in circular orbits is refuted after observing any planet whose orbit is not circular. In logic, such a theory is called incomplete. Conversely, a theory can be too permissive, allowing behaviour that does not appear in the system it is trying to model. The theory of circular orbits can be amended as follows: an orbit is still considered to be circular, but rather than directly orbiting a star, a planet orbits a point. This point may be the center of a star, or it may be another point defined in this manner, so that a planet orbits a series of points whose base point is a star. This theory *can* explain any orbit, and for any given planet, there is list of diameters and frequencies that describe its orbit. The problem is that *any* shape can be drawn with this method, even a square. In logic, a theory with this over-fitting kind of error is called unsound.

A good theory has predictive power. The theory that a planet orbits a star with an elliptical trajectory can also accurately model an orbit, but with this theory, it is possible to ascertain an orbit from only a few points. The theory of orbiting orbits still provides some guarantees about the behaviour of an orbit, such as periodicity, but much is lost. Knowing the exact trajectory of even three quarters of an orbit would give no indication about the last quarter.

Programs can also be viewed through this lens, where the shape of an orbit corresponds to a particular program, and the theory to a language. When a program does not behave as intended, it's often because of a mistake in the implementation – the orbit is simply described with the wrong parameters. Other times, the language itself is too restrictive, like using a theory that can only describe circular orbits. A language with a more powerful set of features may therefore be needed to solve a problem. Because any Turing-complete language is reducible to another, it is often possible to solve a problem using the wrong language for the task, however inelegant the solution.

Sometimes the wrong language is not just inelegant, it's too powerful, when not enough structure is enforced. This is not to say that a well-constrained language will be free of mistakes. Even with an elliptic theory of orbits, it's possible to describe an orbit with the wrong ellipse. But with a good theory, the number of and kind of incorrect orbits is greatly reduced. An untyped language can be best-suited to a problem, and a goto can make a program faster. That many languages have static type systems and lack gotos is precisely because it is worth-while to properly constrain a language. Tailoring a program to a problem can make it easier to reason about about both the program and the problem. Constraint is powerful in and of itself.

## 1.2 Relating Proofs and Programs

Linear Temporal Logic (LTL) is a logic with a first-class notion of time, often in the form of the next-step ($\bigcirc$), always ($\Box$), eventually ($\Diamond$), and until ($\mathcal{U}$) modalities. Beginning with Pnueli [1977], LTL has been used to formally verify systems. It is particularly helpful for concurrent systems, where processes run simultaneously and there is no predetermined total order of program execution. LTL can concisely express temporal properties such as safety - that bad things won't happen, liveness - that good things will, and fairness - that all processes will eventually be served. Other temporal logics can express variations of these guarantees. For example, Computation Tree Logic (CTL) allows quantification over paths and can express certain properties that LTL cannot, although it is unable to express fairness properties. CTL* can express properties of both LTL and CTL. Although CTL* and LTL are both PSPACE-complete [Goranko and Galton, 2015], LTL's formulation is simpler.

The Curry-Howard correspondence relates proof theory and type theory, where logical propositions correspond to types and formal proofs correspond to total functional programs. From a logic, this correspondence can induce a programming language. Linear Temporal Logic corresponds to reactive programming [Jeltsch, 2012] [Jeffrey, 2012]. Reactive programming is used to program graphical user interfaces, games, spreadsheets, and other systems where behaviour of one part of the system is heavily dependent on other parts. When a change is introduced to one part of the system, it causes a reaction that propagates throughout the system.

The relationship between first-order logic and Linear Temporal Logic has been addressed by Kamp [1968], who showed that LTL with the Until and Since operators is as powerful as first order logic with quantification over the integers. Under the Curry-Howard correspondence, predicate logic corresponds to indexed types, a simple kind of dependent type where terms can depend on objects from some index language [Xi and Pfenning, 1999].

## 1.3 Motivation

Multi-core processors are now the norm and distributed systems are commonplace. These systems provide a means to scale computing power, as traditional methods increasingly push up against physical limitations. These systems are also highly concurrent, however, and being able to reason about them rigorously warrants the use of tools like linear temporal logic.

Different logics also provide new directions to expand type-systems via the Curry-Howard correspondence. Rather than reasoning about specific programs, we can enforce a logical property within the type-system of a programming language. This property will therefore hold for all well-typed programs.

This thesis provides an overview of various logical and programming constructs and defines Jackrabbit, a programming language tailored to reactive systems. We also explore the interaction of these constructs within Jackrabbit, which include LTL and first-order logic. Instead of simply modelling LTL, first-order logic can be used to extend LTL so as to specify properties that cannot be expressed in propositional LTL. Jackrabbit's type-system is able to enforce these properties.

## 1.4 Contributions

We introduce Jackrabbit, a reactive programming language with indexed types. The core language [Cave et al., 2014] is based on the modal-$\mu$ calculus and can express interleavable recursive and corecursive types. The indices provide a stronger type system and can express safety guarantees during compilation. Our work uses a next-step modality as a basic construct which helps to provide temporal guarantees, but indices exist irrespective of time. This gives us a simple form of memory and makes it possible to constrain what we are allowed to see depending on what we have seen before.

Rather than iterators and coiterators which are used by Cave et al. [2014], we use Mendler-style recursion [Mendler, 1988], a recursion scheme that ensures terminating functions. This allows the programmer to write recursive functions in a similar way to that of general recursion and uses a simple operational semantics.

The index language is abstract over a domain with decidable equality. In our examples, the index domain consists of natural numbers ($\mathbb{N}$). This extension with indices makes it possible to express tighter constraints. For example, when serving two processes, we can write the safety property that one process is always served more often than another.

These examples, along with the description of Jackrabbit, are presented in chapter 3. A proof of type preservation is given in chapter 4.

### 1.5 Related Work

Esterel [Berry and Cosserat, 1984] and Lustre [Caspi et al., 1987] are both formally specified synchronous programming languages created in France in the 1980's. Our language shares their synchronous notion of time and emphasis on enforcing properties. Neither Esterel nor Lustre are functional, however.

Elliott and Hudak [1997] introduced the functional paradigm to reactive programming and treat time as continuous. Although functional reactive programming lends itself neatly to highly concurrent systems, this occasionally comes with a cost in inefficiency. In particular, space leaks can occur. Krishnaswami et al. [2012] and Krishnaswami [2013] consider the similarity of streams ultrametric spaces to address these space-leaks. They generalize guardedness and are able to place constraints on recursive streams, allowing temporal streams to be defined in terms of themselves while avoiding leaks. Our approach is simpler, but still powerful enough to express interesting properties.

Our treatment of time as a discrete entity is motivated by the next-step modality, $\bigcirc$ [Cave et al., 2014]. This is the base temporal operator in Jackrabbit and tells us that we have access to the type under it at the next time step. Temporal computations are done in lock-step using a synchronous clock, so any two objects under the same number of $\bigcirc$-modalities (say, $\bigcirc\bigcirc A$ and $\bigcirc\bigcirc B$) are accessible at the same moment.

To define LTL's temporal types, the next-step modality is used in tandem with recursive and corecursive types. Recursive types, which correspond to least fixed-points, are common in programming languages and allow us to encode finite inductive structures such as lists and trees. Corecursive types, corresponding to greatest fixed-points, are defined via coinduction, which is the dual to induction in category theory. Corecursive types are used to encode infinite data structures, originally by Hagino [1987]. By interleaving least and greatest fixed points, it's possible to express fairness properties at compile time, ensuring that when serving two processes, each will always eventually be served [Cave et al., 2014]. Unlike Cave et al. [2014], who use iterators and co-iterators, we use Mendler-style (co)recursion Mendler [1988]. This has also been done by Jacob-Rao et al. [2018], without the temporal modality.

# CHAPTER 2
# Predicate Linear Temporal Logic

## 2.1 Linear Temporal Logic

Logic is a tool for reasoning about ideas as abstract as mathematics and as practical as language. Defining different logics allows us to more accurately model specific domains and better reason about particular problems. Linear temporal logic (LTL) is a logic with constructs for making claims about time.

A logic makes a judgment about a proposition. Many logics use the judgment "$A$" to mean "$A$ is true". Linear temporal logic considers $A$ in the context of time, so in LTL, "$A$" means "$A$ is true right now" or "$A$ is true today". This might look like a weaker claim, since the claim "1 + 1 = 2 is true today" seems to draw attention to the fact that it says nothing about "1 + 1 = 2" on any other day. But consider a claim such as "it is sunny", which is true at the time of writing. Through this lens, LTL removes an ambiguity. To do this, LTL provides constructs such as $\Box$, $\Diamond$, and $\mathcal{U}$ that can express temporal claims with finer granularity.

The first "L" in "LTL" is for "linear", meaning that time is treated like a line. It is not possible to reason about different futures. This is in contrast to branching

$$\begin{array}{lrcl} \text{Propositions} & A, B & ::= & \top \mid \bot \mid A \times B \mid A + B \mid A \to B \\ & & \mid & \bigcirc A \mid \Diamond A \mid \Box A \mid A \, \mathcal{U} \, B \mid A \, \hat{\mathcal{U}} \, B \end{array}$$

Figure 2–1: LTL propositions

temporal logics such as Computation Tree logic (CTL) which can quantify over paths into the future. Tomorrow it may or may not rain. If it rains, the grass will grow. LTL can reason about both of those claims, but it cannot make a claim such as "there is a future where the grass will grow". It's still possible to reason about the future before it happens, but there is only one future.

Propositions in linear temporal logic are shown in Figure 2–1. These include the common logical operators of truth, falsehood, conjuction, disjunction, and implication, as well as the temporal operators: the next-step, eventually, always, until, and strong until modalities. We give an overview of these modalities are explained in this chapter and provide more rigorous definitions for their type-theoretic equivalents in the following chapter.

In our presentation of LTL, time is discrete and every moment in time has a successor. Examples have mentioned today and tomorrow, but a moment can be a year, a second, or any other unit of time as long as it is atomic with respect to the logic. This is not always the case in LTL, and will allow us to use tools that deal with discrete entities such as automata. Discrete versions of LTL often have a temporal operator, $\bigcirc A$, which relates a type at the present moment to it's type one step into the future. It can be thought of as a temporal successor operator. If $\bigcirc A$ is true today, then $A$ will be true tomorrow. Unlike natural numbers however, there is no $t = 0$. The notion of time is relative.

$\Box A$ means "always $A$" and is true whenever $A$ is true at every point in time. The claim "1 + 1 = 2" is true in both classical and linear temporal logics. We can strengthen this sentence in LTL to “1 + 1 is always equal to 2”. This is modelled with the $\Box$ modality as “$\Box\ 1 + 1 = 2$”.

It’s also possible to express claims like “some day it will be sunny”, with $\Diamond A$. This is true when there are a finite number of days until $A$ is true. If it is sunny today, we can always deduce that some day it will be sunny. In general, from $\Box A$ we can derive $\Diamond A$. This gives us $\Box A \to \Diamond A$, and in fact there are an infinite number of proofs of this.

The binary operator $A\, \mathcal{U}\ B$ (“$A$ until $B$”) is similar to $\Diamond B$, with the additional requirement that $A$ must be true at every point before that $B$. This is similar to a temporal list whose last element has type $A$. This doesn’t mean that $A$ will ever be true since $B$ might already be true today. That $A$ is true can be enforced with the variant $A\, \hat{\mathcal{U}}\ B$. This ensures that $A$ is true today, and that tomorrow, $A\, \mathcal{U}\ B$.

The modalities $\Box A$ and $\Diamond A$ from linear temporal logic are reminiscent of the modalities “necessarily” and “possibly” from modal logic. We can translate these concepts by relating each possible world in modal logic to a moment in time in temporal logic. The notion “necessarily true” from modal logic, that something is “true in every world”, becomes “true at every point in time” in LTL – always true. Similarly, “possibly true” or “true in some world” becomes “true at some point in time”, or eventually true.

As an example, consider Figure 2–2. At the first time step, we’re in the first state, in which $A$ is true. At every step after that, we’ll be in the second state in which $B$ is true. In the first state, in addition to $A$ being true, we also have $\Diamond B$,

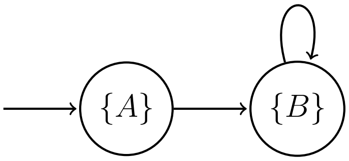

Figure 2–2: $A \times \bigcirc(\Box B)$

$\Diamond\Box B$, $A \,\mathcal{U}\, B$, and $A \,\hat{\mathcal{U}}\, B$. In the second state, $\Box B$, $\bigcirc B$, and $\Box\Box B$ hold, among others.

The more structure we enforce in a given set of objects, the easier they are to reason about. This can come at a cost however, since it's usually necessary to overcompensate and prohibit some objects that do not possess the defective behaviour. Intuitively, it seems that objects at the current time should not be allowed to depend on objects from the future. For arbitrary $A$, it is therefore not possible to write programs of type $\bigcirc A \to A$. This would violate causality, as terms at time $t$ could depend on terms from time $t' > t$ [Jeffrey, 2013]. This is not to say that we cannot write, for example, a term of type $\bigcirc\texttt{bool} \to \texttt{bool}$, just that the boolean produced cannot depend on the input. This still allows constant functions, for example the function that takes a boolean from the future and always returns `true` today.

We also place restrictions on being able to save values for later. Just because it is sunny today does not mean it will be tomorrow. This prohibits proofs of $A \to \bigcirc A$ for arbitrary $A$, although this can be proven for certain instances of $A$. One approach is to treat the past the same as the future, rejecting temporal dependencies altogether. This seems much too restrictive though. The only way to prove that some man will eventually die would be if he were to die that very day. On the other extreme, we could remember everything that has come before just in case past information is

$$\begin{aligned}&\text{save} : \texttt{bool} \to \bigcirc\texttt{bool}\\&\text{save} \equiv \lambda x.\texttt{if}\ x\ \texttt{then}\ \bullet\,\texttt{true}\ \texttt{else}\ \bullet\,\texttt{false}\end{aligned}$$

Figure 2–3: Legal Save

$$\begin{aligned}&\text{save} : \texttt{bool} \to \bigcirc\texttt{bool}\\&\text{save} \equiv \lambda x.\bullet\, x\\&\text{save} \equiv \lambda x.\texttt{if}\ x\ \texttt{then}\ \bullet\, x\ \texttt{else}\ \bullet\, x\end{aligned}$$

Figure 2–4: Illegal Saves

again relevant. This approach has been problematic in implementations of functional reactive programming, even when some things can be forgotten, since space-leaks still arise. In life as well, there are things in the past that have been lost to memory.

By trying to remember too much, it's possible to forget to forget. This introduces space-leaks. Each new time step is therefore a kind of "Groundhog Day", where we forget everything from previous days and the new day seems like the one before. However, in the same way that the protagonist of the film can still apply information he learned before, any object that could be created before can be recreated, as long as it does not contain any variables from a previous day. Anything that cannot be created from scratch, for example, a function provided as an argument, cannot be saved for later. Unlike our restriction for $\bigcirc A \to A$, some information can be transferred to the future. The proposition $A \to \bigcirc A$ need not be constant as we can see in the example for "save" in Figure 2–3. This function returns false at the next step if the input is false, and true at the next step if the input is true. The booleans generated are newly created booleans, however, not copies of the input booleans. The typing rule for $\bigcirc$ moves variables $x$, $a$, and $b$ out of scope when under the $\bullet$, so it is not possible to write "save" as in Figure 2–4.

Classical logic has an axiom called the law of the excluded middle, which presumes that every proposition is either true or false. A proof can categorize a proposition as true or false. However, not every proposition can be put into one of these categories! In 1931, Gödel's incompleteness theorem showed that in any consistent logic, there are statements that are true but are not provable. The version of LTL that we explore is based on intuitionistic logic rather than classical logic. In intuitionistic logic, a true statement is, by definition, a provable statement. Since an inconsistent logic is not of much use, intuitionistic logic does not have the law of the excluded middle. Consider the statement $P + \neg P$. In classical logic, this means "either $P$ is true or $P$ is not true", and this is axiomatically true. In intuitionistic logic, however, this sentence does not mean "either $P$ is provable or $P$ is not provable", but rather "either $P$ is provable or the negation of $P$ is provable". The truth of this statement is incompatible with the incompleteness theorem. For certain statements, however, it is the case that either a sentence or its complement will hold in intuitionistic logic. For example, we can prove that every integer is either even or not even (odd).

The intuitionistic nature of our logic also places a restriction on distributing $\bigcirc$ over sums ('or's). We cannot produce a proof of the proposition $\bigcirc(A + B) \rightarrow \bigcirc A + \bigcirc B$. If we could, we would implicitly gain information about the future since which of the two possibilities hold could be known before either of them are to happen. In other words, given a proof that tomorrow with either be sunny or rainy, it's not possible to prove that tomorrow will be sunny, nor is it possible to prove that tomorrow will be rainy. This only applies in one direction though, so $\bigcirc A + \bigcirc B \rightarrow \bigcirc(A + B)$ is provable. Information is simply lost.

Propositions are needed to state theorems, but the work is still left to prove them. We do not formally provide proof terms for linear temporal logic, nor do we provide the rules for their reduction, although we mention them briefly. We later go into depth defining the terms and operational semantics of Jackrabbit. A proof of a conjunctions is often expressed a pair $(a, b)$, which contains a proof of both the first and the second conjuncts. These proofs can be extracted with `fst` and `snd`. Disjunctions are often introduced with injections, which we introduce in the next section, and eliminated via case analysis. This allows us to prove anything that would follow from either disjunct. An implication $f : A \rightarrow B$ can be introduced with abstraction and eliminated (applying the modus ponens rule) as $f a$, given an antecedent $a : A$. We are interested in defining temporal propositions in terms of least and greatest fixed points. As such, we use induction and coinduction to prove temporal propositions. We define fixed points in the following section.

## 2.2 Fixed Points

Given that it will stop raining tomorrow, it's possible to deduce that it will eventually stop raining. The converse does not hold – if we know that eventually the rain will stop, it could be tomorrow or five years from now. But if the rain eventually stops, then there is some day in the future that will not be rainy. Our version of LTL is based on the propositional $\mu$-calculus [Kozen, 1983], which uses fixed points to relate this set of future days under $\bigcirc$ to the temporal modality "eventually" ($\Diamond$). $\Diamond$ is in fact defined in terms of $\bigcirc$.

A point $x$ that is invariant under a function $f$ is called a fixed point of $f$. It's often possible to find a fixed point for a function $f$ with type $T \to T$ by successively applying $f$ to its output. For example, the function $fx = x/3$ where $x \in \mathbb{N}$ has a fixed point at 0. If we start with any other number and keep on applying $f$, we reach 0. Not all functions have fixed points, though, and even for those that do, this technique does not always work. The boolean "not" function has no fixed point because it alternates between two values. For the function $fx = -1 \times x$ for $x \in \mathbb{Z}$, repeatedly applying $f$ to any input value but 0 will never produce the fixed point $x = 0$. For other functions, such as $fx = x + 1$ where $x \in \mathbb{Z}$, there is no fixed point because there is no integer such that $fx = x + 1$.

To ensure that types defined with fixed points are meaningful, we enforce some structural requirements. We take the fixed points of monotone functions in a complete partial order (CPO). A monotone function $f$ is a function that preserves order: $x \leq y \Rightarrow fx \leq fy$. This rules out functions that oscillate, such as "not" and $fx = -1 \times x$.

A partial order is a set along with "$\leq$", a reflexive, anti-symmetric, and transitive ordering. That is, for all elements $x$, $y$, and $z$, the following properties hold:

- $x \leq x$
- if $x \leq y$ and $y \leq x$ then $x = y$
- if $x \leq y$ and $y \leq z$, then $x \leq z$

Not all pairs of elements need be comparable under this ordering though, it's possible that neither $x \leq y$ nor $y \leq x$.

In a complete partial order, every subset must have a least upper bound and a greatest lower bound. A point $u$ is an upper bound of $S$ whenever $s \leq u$ for every point $s \in S$. Let $U$ be the set of upper bounds of $S$. A point $j$ is a least upper bound of $S$ when $j \in U$ and for every upper bound $u \in U$, $j \leq u$. A greatest lower bound is defined similarly. $\mathbb{N}$ and $\mathbb{Z}$ with the $\leq$ ordering on integers are not CPOs because there is no least upper bound.

### 2.2.1 Least Fixed Points

Least fixed points, written $\mu$, are used to define inductive structures. These are unbounded but finite, and include lists, trees, $\Diamond$, and natural numbers. A point $x$ is a least fixed point of $f$ when it is a fixed point, and for every other fixed point $y$ of $f$, $x \leq y$. For consistency with types, we will use capital letters for functions and points after this point.

**Theorem 1. Kleene Fixed Point Theorem** *Let $(\mathcal{L}, \leq)$ be a complete partial order and $F : \mathcal{L} \to \mathcal{L}$ be a monotone function. Then $F$ has a least-fixed point, given by the supremum of the ascending Kleene chain of $F$.*

To define natural numbers, we take the partial order $(\mathcal{L}, \subseteq)$, where $\mathcal{L}$ is a set of terms and $\subseteq$ is the subset ordering. The function $F$ is defined as $F\ X = 1 + X$. 1 is the set containing () (unit) and + corresponds to tagged set union ($\cup$), tagging the left and right elements with `inl` and `inr` respectively.

Now, we want to find the least fixed point of this equation. Since $F$ is monotone ($X \subseteq 1 + X$), we apply the Kleene fixed point theorem. To find the ascending Kleene chain of $F$, we start with $X = \emptyset$ and build up:

$$\begin{array}{lcl}
F\ \emptyset & = & \{\texttt{inl}\ ()\} \\
F(F\ \emptyset) & = & \{\texttt{inl}\ ()\} \cup \{\texttt{inr}\ (\texttt{inl}\ ()) \\
F(F(F\ \emptyset)) & = & \{\texttt{inl}\ ()\} \cup \{\texttt{inr}\ (\texttt{inl}\ ())\} \cup \{\texttt{inr}\ (\texttt{inr}\ (\texttt{inl}\ ()))\} \\
\ldots & &
\end{array}$$

The least fixed point of $F$, written $\mu X.1 + X$, is the least upper bound of this chain, the smallest set such that $F\ X = X$. This set is isomorphic to the set of natural numbers, where $\texttt{inl}\ () \cong 1, \texttt{inr}\ (\texttt{inl}\ ()) \cong 2, \texttt{inr}\ (\texttt{inr}\ (\texttt{inl}\ ())) \cong 3$, etc.

It's also possible to define natural numbers purely in terms of sets. Set union doesn't distinguish left from right, however, so sets representing natural numbers must be distinguished by what they contain which can be harder to read. The number 4, for example, is represented as the set containing 0, 1, 2, and 3: $\{\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}, \{\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}\}\}$

### 2.2.2 Greatest Fixed Points

Dually, we can define co-natural numbers co-inductively. Rather than taking the ascending Kleene chain of $F$, starting with Ø, we take the descending Kleene chain of $F$, starting with $\Omega$, the set of all terms. Instead of adding to this, we refine:

$$\begin{array}{lcl}
\Omega & = & \{x \mid x \in\ \Omega\} \\
F\ \Omega & = & \{\texttt{inl}\ ()\} \cup \{\texttt{inr}\ x \mid x \in\ \Omega\} \\
F(F\ \Omega) & = & \{\texttt{inl}\ ()\} \cup \{\texttt{inr}\ (\texttt{inl}\ ())\} \cup \{\texttt{inr}\ (\texttt{inr}\ x) \mid x \in\ \Omega\} \\
\ldots & &
\end{array}$$

The greatest fixed point, $\nu X.1 + X$, is the greatest lower bound of this chain, the biggest set such that $F\ X = X$. This set includes the "point at infinity", `inr` (`inr` (`inr` $\cdots$)), since at no point is it removed. In the least fixed point example, this term is never added. We still use $\cup$ because it corresponds to the the $+$ operation of $F$. The intersection happens between iterations: $F\ \Omega = \Omega \cap (1 + \Omega)$. This is in contrast to union for the least fixed point, where $F\ \emptyset = \emptyset \cup (1 + \emptyset)$.

### 2.2.3 Defining Types

The power of linear temporal logic comes from the temporal operators always ($\Box A$), eventually ($\Diamond A$), until ($A\ \mathcal{U}\ B$), and strong until ($A\ \hat{\mathcal{U}}\ B$). They are defined in terms of fixed points and the $\bigcirc$-modality as follows:

$$\begin{array}{rcll} \Box A & \equiv & \nu Z.A \times \bigcirc Z & \text{(Always)} \\ \Diamond A & \equiv & \mu Z.A + \bigcirc Z & \text{(Eventually)} \\ A\ \mathcal{U}\ B & \equiv & \mu Z.B + A \times \bigcirc Z & \text{(Until)} \\ A\ \hat{\mathcal{U}}\ B & \equiv & A \times \bigcirc(A\ \hat{\mathcal{U}}\ B) & \text{(Strong Until)} \end{array}$$

The temporal proposition $\Diamond A$ is defined as the least fixed point of the function $FX = A + \bigcirc X$. With natural numbers, although the set is infinite, any given number has a finite distance from 0. The same is true of $\Diamond$, which makes it possible to ensure that if $\Diamond A$, then $A$ will happen some day.

Always $A$ can be interpreted as a stream of $a$'s. It is defined with the greatest fixed point $\nu$ because cofinite nature of the claim that $A$ is true at every point in time. The type $\Box A \to A$ is therefore inhabited by the head function on streams. Because these streams are temporal, however, it is not in general possible to write a function of type $A \to \Box A$. This is for the same reason that $A \to \bigcirc A$ isn't possible, as they both attempt save an $A$ for later. The converse holds for $\Diamond A$. $A \to \Diamond A$ is inhabited but $\Diamond A \to A$ is not. $\Diamond A \to A$ entails potentially importing an $A$ from the future and therefore isn't allowed for the same reason as $\bigcirc A \to A$. Unfortunately, $\Diamond$ does not form a monad in Jackrabbit. $A \to \Diamond A$ can be seen as lifting $A$, and it's possible to derive $\Diamond\Diamond A \to \Diamond A$, but it's not always possible to map over $\Diamond$. Given a function $f : A \to B$ and a value $v : \Diamond A$, the function may be forgotten by the time it can be applied.

### 2.3 Predicates

Abstraction is the power to see the similarities among different things. It ignores information that is irrelevant for a given use so that different objects can be seen in the same light. We treated fixed points as a means of abstraction relative to simple types. Instead of considering $\bigcirc A$ and $\bigcirc \bigcirc A$ as different propositions, fixed points allow them to be seen as different instances of the same proposition, $\Diamond A$ (or $\mu Z.A + \bigcirc Z$).

Sometimes, however, the differentiating information is useful. Rather than erasing information altogether, predicates allow us to decouple this information from a proposition. We can index $\Diamond A$ by a natural number representing its size. $\Diamond^1 A$ means that $A$ will be true at the next step, $\Diamond^2 A$ that $A$ will be true in two steps. This indexed $\Diamond$ is called a predicate. Indexing a proposition with a concrete object does not add anything itself, but it lays the groundwork for abstract indices. This will allow us to place tighter constraints on types.

A classic example of predicate logic is a syllogism about Socrates' mortality. The third sentence follows from the first two: "All men are mortal. Socrates is a man.

$$\Pi x.Mx \rightarrow \Diamond Dx,\ Ms \vdash \Diamond Ds$$

Figure 2–5: A Derivation of Socrates's Death

Therefore Socrates is mortal." In propositional logic, each of these sentences can be represented, but they have no bearing on one another. They are atomic sentences – they cannot be further divided. One attempt to derive Socrates' mortality is to change the sentences to the following: "If Socrates is a man, then Socrates is mortal. Socrates is a man. Therefore Socrates is mortal." But if we learn that Plato is also a man, we cannot derive Plato's mortality. We would also have to know that "If Plato is a man, then Plato is mortal".

In predicate logic, we are able to split each of these sentences up so that we can model their internal structure. This uncoupling allows us to make universal and existential claims, not about a particular person, but rather truths about classes of objects. "Socrates is a man." is broken into two parts: $M$, the property of being a man, and Socrates, the object $s$. We write this $Ms$. The sentence "All men are mortal" is a statement about men in the abstract. We write this as $\Pi x.Mx \rightarrow Dx$: "for all objects $x$, if $x$ has the property of being a man then $x$ has the property of being a thing that will die. To derive the conclusion "Therefore Socrates is mortal", we use a rule that allows us to instantiate universal claims. If some thing is true of all objects, then it is true of a particular object. This gives us the sentence $Ms \rightarrow Ds$: "If Socrates is a man, then Socrates is mortal.", where $s$ = Socrates, at which point we can derive the conclusion. This is a temporal claim, however! We can express even more structure using indexed-LTL in Figure 2–5, where $Da$ means "$a$ is dead" rather than "$a$ will die".

Quantifiers can be used to model the temporal modalities $\Box$ and $\Diamond$. The index domain consists of present and future moments in time, which can be represented by natural numbers. We can then define $\Box A$ as $\Pi x.Ax$. $\Diamond A$ is defined with the existential quantifier, $\Sigma$. Whereas $\Pi$ is used to make claims about entire classes of objects, $\Sigma$ is used to specify that there is at least one thing that has some property. Since "things" here are moments in time, $\Sigma x.Ax$, means that "there is a moment in time where $A$ is true". That is, $\Diamond A$. Jackrabbit uses indexed types and temporal modalities differently and therefore does not blend them together.

It's often useful to reason about the relationship between two objects. We use equality (=) to express that two objects are equal. Along with quantifiers, this makes it possible to express that there's at most one thing with some property $P$: $\Pi x.\Pi y.Px \times Py \to x = y$ or that there are at least two things: $\Sigma x.\Sigma y.Px \times Py \times x \neq y$.

# CHAPTER 3
# The Jackrabbit Language

The Curry-Howard correspondence relates logical propositions to program types. In the previous chapter, we focused on logic. Here, we focus on programs. Jackrabbit is defined in this chapter, although we also introduce a few programming concepts absent from Jackrabbit so as to better explain design choices. Most of the examples are towards the end of the chapter, after having provided an explanation of Jackrabbit's constructs. To give a taste of Jackrabbit now, we provide a few examples using the modalities introduced in the previous chapter. We also use booleans in these examples. Since a boolean can be true or false, it can be defined in Jackrabbit with a sum type as in Figure 3–1.

**Example 1** (Two-steps)**.** Recall that the eventually modality $\Diamond A$ is defined as $\mu Z.A + \bigcirc Z$. We define true two steps from now in Figure 3–2, which has type $\Diamond\texttt{bool}$. Seeing as $\Diamond\texttt{bool}$ is syntactic sugar, we expand the full type to $\mu Z.(1 + 1) + \bigcirc Z$ in this example for clarity. The $\texttt{inj}$ is used to create a recursive type, and replaces the type variable $Z$ by $\mu Z.A + \bigcirc Z$. Without the $\texttt{inj}$ terms, we would have $\texttt{inr}\ \bullet(\texttt{inr}\ \bullet(\texttt{inl true}))$ of type $\texttt{bool} + \bigcirc(\texttt{bool} + \bigcirc(\texttt{bool} + \bigcirc(\mu Z.\texttt{bool} + \bigcirc Z)))$.

$$\begin{aligned}
\texttt{bool} &\equiv 1 + 1 \\
\texttt{false} &\equiv \texttt{inl}\ () \\
\texttt{true} &\equiv \texttt{inr}\ ()
\end{aligned}$$

Figure 3–1: Booleans

$$\begin{array}{l} \mathit{TwoSteps} : \mu Z.(1+1) + \bigcirc Z \\ \mathit{TwoSteps} \equiv \texttt{inj}\ (\texttt{inr}\ (\bullet\texttt{inj}\ (\texttt{inr}\ (\bullet\texttt{inj}\ (\texttt{inl}\ (\texttt{inr}\ ()))))))) \end{array}$$

Figure 3–2: Two-steps

**Example 2** (Stream application)**.** Corecursion allows us to handle streams, which correspond to the $\Box$ modality. Here, we create a stream with the Sisyphean task of continually applying a function to a value, each of which is passed in via an input stream. This example, shown in Figure 3–3, is modified from Cave et al. [2014]. *Apply* is a curried function that takes two parameters. We uncurry these parameters and pass them as a single tuple $(fs, as)$ to the `corec` term, whose signature is $(\Box(A \to B),\ \Box A)\ \to\ \Box B$. The variable $r$ gives us access to the object we pass in as $(fs, as)$ and $f$, unrelated to $fs$, is a function used to make the corecursive call. *hd* and *tl* are simply syntactic sugar for `fst` and `snd`, used here to clarify that we're acting on a stream. The `let`'s take a next step term of type $\bigcirc A$ and give us access to the term of type $A$, binding it here to the variables $fs'$ and $as'$. We then reconstruct a new stream whose head is the head of $fs$ applied to the head of $as$. In the tail of the stream, we apply $f$ to $fs'$ and $as'$. This $f$ and its recursive equivalent live in the eternal context, so unlike most variables, they can be used under a $\bullet$ term. We explain this in more depth later.

**Example 3** (In-Order Buffer)**.** In Jackrabbit, we'll also use indices in types, which correspond to logical predicates. We define vectors soon, but to give an idea of how indices can be used, we show the type of a simple buffer that reads and then forwards a value. Simply expressing that a value read will be sent immediately

$$\begin{array}{l}
\mathit{Apply} : \Box(A \to B) \to \Box A \to \Box B \\
\mathit{Apply} \equiv \lambda fs.\lambda as. \\
\texttt{corec}^{(\Box(A\to B),\ \Box A)\ \to\ \Box B}(rpf. \\
\qquad \texttt{let } \bullet fs' = tl\ (\texttt{fst}\ r)\ \texttt{in} \\
\qquad \texttt{let } \bullet as' = tl\ (\texttt{snd}\ r)\ \texttt{in} \\
\qquad ((hd\ (\texttt{fst}\ r))(hd\ (\texttt{snd}\ r)), \bullet f(fs', as')) \\
)(fs, as)
\end{array}$$

Figure 3–3: Stream Apply

| | | | |
|---|---|---|---|
| Kinds | $K$ | $::=$ | $\texttt{type} \mid \Pi u{:}U.K$ |
| Index type | $U$ | $::=$ | $\mathbb{N}$ |
| Index objects | $C$ | $::=$ | $u \mid \text{zero} \mid \text{suc } C$ |
| Type families | $R$ | $::=$ | $Z \mid \mu Z.\Lambda\overrightarrow{u{:}U}.A \mid \nu Z.\Lambda\overrightarrow{u{:}U}.A$ |
| Types | $A, B, S, T$ | $::=$ | $1 \mid A \to B \mid A \times B \mid A + B \mid \bigcirc A$ |
| | | $\mid$ | $R\ \vec{C} \mid \Pi u{:}U.A \mid \Sigma u{:}U.A \mid C_1 = C_2 \mid R$ |

Figure 3–4: Types, Kinds, and Indices

might be too strong a constraint. On the other hand, expressing that values received will eventually be sent still allows for out-of-order transmission. The following type requires that values read will be sent in the same order as they have been received. Here, we read and write values from the index language. $R$ and $S$ are the predicates for receiving and sending, respectively, and $\Pi$ takes as input two natural number messages, first $m$ and then $n$, and eventually sends $m$ followed by $n$.

Buffer $\equiv \Pi m{:}\mathbb{N}.\Pi n{:}\mathbb{N}.(Rm \times \Diamond Rn) \to \Diamond(Sm \times \Diamond Sn)$

## 3.1 Syntax of Types

Semantically, a proof is different than a program. A proof is concerned with the truth-value of a proposition, whereas a program is used to solve a problem via computation. Syntactically, proofs are often different as well, which can help to illustrate the difference in meaning. For simplicity, the previous section uses the syntactic constructs of Jackrabbit for proofs rather than a syntax more common to logic. For example, logical conjunction is often written "&" or "$\wedge$", but conjunction in the previous section is written "$\times$", as are products in Jackrabbit.

Jackrabbit includes functions $A \to B$ which correspond to logical implication, products $A \times B$ which correspond to logical conjunction, and sum types $A + B$, corresponding to logical disjunction. The unit type (1) corresponds to the logical value "true". The syntax of each of these is the same as that of its logical counterpart, along with the next-step modality $\bigcirc A$, $\Sigma$ types which correspond to existential quantification $\Sigma x.Px$, $\Pi$ types which correspond to universal quantification $\Pi x.Px$, and index equality $(C_1 = C_2)$. Logically, $=$ often refers to index identity. Two different things can be equal in some respect without being identical. Since we are only comparing index objects here, we simply use equality.

The proposition $\Sigma x.Px$ is made up of a few different parts: the quantification, a variable over which we quantify, the predicate, and the index $x$. We usually use $u$ as an index variable rather than $x$ and annotate its type: $u{:}U$. In addition to variables, the index domain consists of natural numbers, defined inductively. This is a simple index language but is sufficient for our purposes.

In the example $\Sigma x.Px$, $P$ is a predicate and only becomes a proposition once it has taken all of its indices. Logical predicates translate to $\Pi$-kinded types in Jackrabbit. We extend (co)recursive types to (co)recursive types with $\Lambda u$, making it possible to embed indices into inductive and coinductive objects. With this extension,

| | | | |
|---|---|---|---|
| Variable contexts | $\Theta, \Gamma, \Omega$ | $::=$ | $\cdot \mid \Gamma, x{:}T$ |
| Eternal context | $\Delta$ | $::=$ | $\Phi; \Psi \mid \Phi; \Psi; \Omega$ |
| Type variable context | $\Phi$ | $::=$ | $\cdot \mid \Phi, Z{:}K$ |
| Index context | $\Psi$ | $::=$ | $\cdot \mid \Psi, u{:}U$ |

Figure 3–5: Contexts

it becomes necessary to ensure not only that terms are well-typed, but that types are well-kinded.

Kinds create a kind of inductive type for types and type constructors. The syntax of our types is shown in Figure 3–4. All types in non-indexed (and non-dependent) languages have kind `type`, but we also allow for objects that are waiting to take index objects. To better illustrate this, we call these type families ($R$). Type families can have kind $\Pi u{:}U.K$, where $u$ is an index variable from index domain $U$ and $K$ is an inductively defined kind. They include recursive objects, corecursive objects, and type variables. Although $R$ without applying any indices is included as a type here, the requirement that type families must take all of their indices is enforced by the kinding rules.

Vector arrows ($\vec{\ }$) denote lists of indices. For example, $\overrightarrow{u{:}U}$ is a list of index objects $u_i$, each of which has index type $U_i$. For types that are only defined with a single index, we use vector notation as syntactic sugar to extend the definition to multiple indices. For example, $\mu\Lambda\overrightarrow{u{:}U}.A : \Pi\overrightarrow{u{:}U}.K$ is equivalent to $\mu\Lambda u_1{:}U_1, \cdots, \Lambda u_n{:}U_n.A : \Pi u_1{:}U_1, \cdots, \Pi u_n{:}U_n.K$. For recursive types that take no indices, we leave out the $\Lambda$ and kind annotation, since $K$ must have kind `type`. $\mu Z.A + \bigcirc Z$ is therefore equivalent to $\mu Z.\Lambda.A + \bigcirc Z$.

### 3.2 Contexts

We use contexts $\Gamma$, $\Theta$, $\Omega$, $\Phi$, and $\Psi$ to store three kinds of variables. $\Gamma$, $\Theta$, and $\Omega$ keep track of term variables $x$. The typing rules make it possible to move variables between these contexts, enforcing temporal constraints. $\Gamma$ contains variables at the current time step and $\Theta$ contains variables that may be used in the future, replacing the variables in $\Gamma$. Term variables can also be stored in $\Omega$, which is invariant when stepping under a $\bullet$. $\Phi$ stores type variables, and $\Psi$ stores index variables which have type $\mathbb{N}$ in our examples.

We join contexts $\Phi$, $\Psi$, and $\Omega$ into the eternal context $\Delta = (\Phi; \Psi; \Omega)$, and note that objects in $\Delta$ do not depend on time. Since $\Omega$ is only required for the recursive and corecursive terms, we can also write the extended context leaving out $\Omega$, as $\Delta = (\Phi; \Psi)$. Objects in $\Gamma$ and $\Theta$ depend on both the current time step and, potentially, objects in $\Delta$. Each variable $Z$ in $\Phi$ is annotated with its kind $K$, which indicates the number and kind of indices that $Z$ is waiting to take. The kind also specifies the domain $U$ of each index. In examples where index variables do not depend on other index variables, we can write $U \to K$ for $\Pi u{:}U.K$. We also write $\Delta, u{:}U$ as shorthand for $\Phi; (\Psi, u{:}U)$ and $\Delta, Z{:}K$ for $(\Phi, Z{:}K); \Psi$.

### 3.3 Kinding Rules

The kinding rules presented in Figure 3–6 are based on those from Cave and Pientka [2012], with the addition of the circle modality [Cave et al., 2014] and corecursive types. A kind is well-formed according to the judgment $\Delta \vdash T : K$. This states that type $T$ has kind $K$ in context $\Delta$. Similarly, $\Delta \vdash C : U$ means that index

$$\frac{\Delta \vdash \overrightarrow{U : \texttt{mtype}} \qquad \Delta, \overrightarrow{u{:}U}, Z{:}\Pi\overrightarrow{u{:}U}.K \vdash T : K}{\Delta \vdash \mu Z.\Lambda\overrightarrow{u{:}U}.T : \Pi\overrightarrow{u{:}U}.K} \quad \frac{Z{:}K \in \Delta}{\Delta \vdash Z{:}K}$$

$$\frac{\Delta \vdash \overrightarrow{U : \texttt{mtype}} \qquad \Delta, \overrightarrow{u{:}U}, Z{:}\Pi\overrightarrow{u{:}U}.K \vdash T : K}{\Delta \vdash \nu Z.\Lambda\overrightarrow{u{:}U}.T : \Pi\overrightarrow{u{:}U}.K} \quad \frac{\Delta \vdash A : \texttt{type}}{\Delta \vdash \bigcirc A : \texttt{type}}$$

$$\frac{\Delta \vdash A : \texttt{type} \qquad \Delta \vdash B : \texttt{type} \qquad * \in \{\to, \times, +\}}{\Delta \vdash A * B : \texttt{type}}$$

$$\frac{\Delta \vdash C_0 : U \qquad \Delta \vdash C_1 : U}{\Delta \vdash C_0 = C_1 : \texttt{type}} \quad \frac{\Delta \vdash \overrightarrow{U : \texttt{mtype}} \qquad \Delta, \overrightarrow{u{:}U} \vdash T : \texttt{type}}{\Delta \vdash \Sigma\overrightarrow{u{:}U}.T : \texttt{type}}$$

$$\frac{\overrightarrow{\Delta \vdash U : \texttt{mtype}} \qquad \Delta, \overrightarrow{u{:}U} \vdash T : \texttt{type}}{\Delta \vdash \Pi\overrightarrow{u{:}U}.T : \texttt{type}} \quad \frac{\Delta \vdash R : \Pi\overrightarrow{u{:}U}.K \qquad \Delta \vdash \vec{C} : \vec{U}}{\Delta \vdash R\ \vec{C} : \{\vec{C}/\vec{u}\}K}$$

Figure 3–6: Kinding Rules

$C$ is well-formed and has index type $U$, and $\Delta \vdash U : \texttt{mtype}$ means that index type $U$ is a well-formed meta-type.

Each of $\bigcirc A$, $\to$, $\times$, and $+$ is well-kinded if each of its subtypes is too. We note that with $\to$, there is no restriction on type variables in the function domain, since our recursion scheme ensures that recursion with these types is well-behaved.

The least and greatest fixed points are well-kinded when given a type variable and indices that can produce a type $T$ of kind $K$. This parallels the typing rule for function abstraction with both a type variable and a vector of index variables. Applying indices to $\Pi$-kinded types can be done as $R\ \vec{C}$, which is similar to function application. $\{\vec{C}/\vec{u}\}K$ means that occurrences of the variable $u$ in $K$ are replaced by $\vec{C}$. This is further explained in the section on substitution.

Kinding for Π and Σ-types is also similar to function abstraction with a vector of index variables. Given a context containing a vector of well-kinded index variables, we can create a type that does not depend on these. It's also important to note that the Π-kind on the right of a ":" and the Π-type on the left of a ":" are unrelated. Although they both take indices, they exist on different levels.

## 3.4 (Co)Recursion

A type $A$ has introduction rules which are used to create terms of type $A$, and elimination rules, used to produce a term of type $B$ from a term of type $A$. For some types, there is not a lot of choice to be made about the introduction and elimination forms. For example, the product type $A \times B$ is usually introduced with a pair, $(a, b)$. A term $M : A \times B$ can be eliminated with the projections `fst` $M$ and `snd` $M$, or else with a `let` term that binds both elements to variables, but there are only so many ways to put two objects into a pair and to take them out.

When it comes to recursive types, there's much more variety. Full recursion, although simple and powerful, can produce programs that never end. We introduce a few alternatives and examine their strengths and weaknesses before presenting a simplified version of the recursion scheme used in Jackrabbit.

### 3.4.1 General Recursion

Recall that a fixed point $X$ of a function $F$ is a point such that $FX = X$. Fixed points can be used to define the operational semantics of terms with recursive types. The following presentation of general recursion, from Pierce [2002], provides an operator `fix` which takes a function $\lambda f.M : T \to T$ and naively unrolls the function definition during evaluation, replacing $f$ by itself. This provides for a simple

$$\frac{\Gamma \vdash M : T \to T}{\Gamma \vdash \texttt{fix}\ M : T}$$

Figure 3–7: Typing Rule for General Recursion

$$\frac{\Gamma \vdash M : [\![\mu X.S/X]\!]S}{\Gamma \vdash \texttt{inj}\ M : \mu X.S}\ \mu I$$

Figure 3–8: $\mu$ Introduction

and intuitive stepping rule:

$$\texttt{fix}(\lambda f.M) \longrightarrow [\texttt{fix}(\lambda f.M)/f]M$$

Since no base case is mandated, it is left up to the programmer to ensure that the recursion is well-founded. Consider the `not` function over booleans. This can be written in pseudocode as $\lambda x.$`if` $x$ `then false else true`. The `fix` operator turns this `bool` → `bool` function into a term of type `bool`. But this boolean is neither true nor false, and the program will never end. General recursion therefore fails to ensure that recursive functions terminate.

### 3.4.2 (Co)Iterators

Sometimes non-terminating programs are useful. These are possible to write via corecursion and with `fix`, but no guarantees are made about normalization and totality of functions. This makes it harder to reason about the system. One way to restore normalization is with iterators (Figure 3–10) and coiterators (Figure 3–12).

$$\frac{\Gamma \vdash M : \nu X.S}{\Gamma \vdash \texttt{out}\ M : [\![\nu X.S/X]\!]S}\ \nu E$$

Figure 3–9: $\nu$ Elimination

$$\frac{x : [\![T/X]\!]S \vdash M : T \qquad \Gamma \vdash N : \mu X.S}{\Gamma \vdash \texttt{iter}^{X.S}(x.M)N : T}$$

Figure 3–10: Typing Rule for Iterators

```
length ≡ $\lambda l.$iter$^{X.1+A\times X}$
           ($x$.case $x$ of
             | inl $a \mapsto$ inj (inl ())          - - Empty list
             | inr $b \mapsto$ inj (inr (snd $b$))   - - snd $b$ : nat
           ) $l$
```

Figure 3–11: Length with Iterators

Iterators can only recurse over inductive objects, which are well-founded and will therefore terminate. Unfolding of cofinite objects is done with `out` and coiterators only compute to as many positions as are requested; it is not possible to retrieve an infinite object before doing anything. For example, the corecursive analog of $\texttt{fix}(\lambda x.\ x)$ is inhabited: $\texttt{coit}^{X.X}(x.x)() : \nu X.X$, which is a stream with no data. Unlike $\texttt{fix}(\lambda x.\ x)$, the program $\texttt{coit}^{X.X}(x.x)()$ will not step until a request is made to do so with `out`.

The body of an iterator only has access to the variable $x$, the recursive object. At each step as the iterator traverses the inductive object, the size of the body decrements. Restricting the context to only contain $x$ ensures that variables aren't carried forward unnecessarily. The object over which we are inducting does not have to be a closed type, and so the $\Gamma$ from the right axiom can be carried over. This is useful for functions like $\lambda x.\texttt{iter}(M)\ x$.

In Figure 3–11 the length of a list $l$ is computed. A list can either be empty (the `inl` case), or have a head and a tail (the `inr` case). Intuitively then, $b$ looks

$$\frac{x : S \vdash M : [\![S/X]\!]T \qquad \Gamma \vdash N : S}{\Gamma \vdash \texttt{coit}^{X.T}(x.M)N : \nu X.T}$$

Figure 3–12: Typing Rule for Coiterators

```
conat ≡ $\lambda n.$coit$^{X.1+X}$
          ($x$.case $x$ of
             | inl $a \mapsto$ inl ()          - - no inj or out
             | inr $b \mapsto$ inr $b$
          ) $n$
```

Figure 3–13: Converting a Nat to a Conat

like a pair of a list head and tail. However, the typing rule (Figure 3–10) replaces the input type variable with the output type, so the tail of $b$ actually has type nat. “- -” begins a line comment and is purely for the reader.

The operational semantics of iterators are not as elegant as those of general recursion. Cave et al. [2014] address the semantics by introducing a new term, called `map`, whose stepping rule introduces and eliminates every type while expanding recursive types before the iterator is applied. This introduces an extra layer, since `map` is a term that is never directly used by the programmer. This can also be inefficient in programs that don’t need to traverse the entire inductive object, such as taking the tail of a list. The inductive object must be reconstructed, which necessitates a non-trivial program for a trivial task and takes linear time in the size of the inductive object.

$$\frac{x : [\![T \times \mu X.S/X]\!]S \vdash M : T \qquad \Gamma \vdash N : \mu X.S}{\Gamma \vdash \texttt{prim}^{X.S}(x.M)N : T}$$

Figure 3–14: Typing Rule for Primitive Recursion

```
pred ≡ $\lambda n.$prim$^{X.1+X}$
          ($x$.case $x$ of
             | inl $a \mapsto$ inj (inl ())
             | inr $b \mapsto$ snd $b$          -- snd $b : \mu X.1 + X$
           ) $n$
```

Figure 3–15: Primitive Recursive Predecessor

### 3.4.3 Primitive Recursion

To address the issue of recreating the inductive object, we turn to primitive recursion, whose typing rule is shown in Figure 3–14. Primitive recursion gives us access to two objects, the object being recursed over and the object being created. The former is immediately accessible, which lets us simplify functions like the tail function and only takes constant time. However, this makes the typing even less intuitive, since our recursive object is now a pair. Even though we now need to use the first and second projections, no pairs show up anywhere in the input.

In Figures 3–15 and 3–16, `fst` $b$ and `snd` $b$ have the same type, since the input type is the same as the output type. Importantly, the operational semantics differ. If we were to replace `snd` with `fst` in Figure 3–15, our function would be the constant

```
copy ≡ $\lambda n.$prim$^{X.1+X}$
          ($x$.case $x$ of
             | inl $a \mapsto$ inj (inl ())
             | inr $b \mapsto$ inj (inr (fst $b$)) -- fst $b : \mu X.1 + X$
           ) $n$
```

Figure 3–16: Primitive Recursive Copy

$$\frac{\cdot; x : [\![T/X]\!]S \vdash M : T \qquad \Theta; \Gamma \vdash N : \mu X.S}{\Theta; \Gamma \vdash \mathtt{iter}^{X.S}(x.M)N : T}$$

Figure 3–17: Temporal Iterator

function 0, since each recursive call would peel off a successor. In Figure 3–16, replacing `fst` by `snd` would still yield the copy function, but would only take constant time.

### 3.4.4 Mendler-style Recursion

Although it improves the performance of certain functions, primitive recursion does not resolve any issues pertaining to the operational semantics and uses a less intuitive typing rule. Mendler-style recursion [Mendler, 1988] in Figure 3–19, allows the programmer to write functions the same way as with general recursion, by giving the programmer access to a function named $f$. Because $f$ can only be applied to well-founded inductive objects, termination is restored since the recursive call can only be made on a smaller object. The operational semantics are also similar to general recursion.

Another benefit of Mendler-style recursion is that it doesn't require that type variables only occur in positive positions, that is, on the left side of an even number of $\rightarrow$'s. With this restriction, types such as $\mu Z.(Z \rightarrow A) \rightarrow Z$ are allowed but $\nu Z.Z \rightarrow Z$ and $\mu Z.Z \rightarrow A$ are not. Proving normalization in languages with iteration can be done using a monotonicity witness, that $\forall X.\forall Y.(X \rightarrow Y) \rightarrow [\![Y/X]\!]A$, where $\forall X$ means for all objects $X$. This ensures that type variable renaming is a finite process. Mendler-style recursion does not require a monotonicity witness because its interpretation is the least fixed point of a monotone operator [Matthes, 1999] .

$$\frac{\cdot; x : [\![T \times \mu X.S/X]\!]S \vdash M : T \qquad \Theta; \Gamma \vdash N : \mu X.S}{\Theta; \Gamma \vdash \texttt{prim}^{X.S}(x.M)N : T}$$

Figure 3–18: Temporal Typing Rule for Primitive Recursion

$$\frac{f : Y \to T, r : [\![Y/X]\!]S \vdash M : T \qquad \Gamma \vdash N : \mu X.S}{\Gamma \vdash \texttt{rec}^{X.S}(rf.M)\ N\ \ : T}\ \mu E$$

Figure 3–19: Typing for Mendler-style Recursion

The typing rule presented loses the benefit of quick access to the inductive object offered by primitive recursion. This can be restored by adding the function $p$ as in Figure 3–21, which corresponds to the second element of the pair in the primitive recursive approach. Mendler-style corecursion, shown in Figure 3–22, is done similarly. We explain these rules further in the typing section, as well as the full rules which contain indices. The rules presented here also replace $X$ by the type variable $Y$. This makes explicit the fact that applying $f$ or $p$ are the only things that can be done. However, $p$ is replaced by the identity function under type substitution during evaluation. We can refrain from substituting $Y$ and using $p$ as a function, and simply use $r$ when we wish to return the inductive object.

```
length ≡ λl.rec^{X.1+A×X}
            (rf.case r of
               | inl a ↦ inj (inl ())          - - nil
               | inr b ↦ inj (inr (f b))
            ) l
```

Figure 3–20: Length with Mendler-style recursion

$$\begin{array}{c}\mathcal{P} \equiv Y \to \mu X.S \\ \mathcal{F} \equiv Y \to T \\ \dfrac{f : \mathcal{F}, p : \mathcal{P}, r : [\![Y/X]\!]S \vdash M : T \qquad \Gamma \vdash N : \mu X.S}{\Gamma \vdash \mathtt{rec}^{X.S}(rpf.M)\ N\ \ : T}\,\mu E\end{array}$$

Figure 3–21: Mendler-style Recursion with $p$ (and without indices)

$$\dfrac{f : T \to Y, r : S \vdash M : [\![Y/X]\!]T \qquad \Gamma \vdash N : S}{\Gamma \vdash \mathtt{corec}^{X.T}(rf.M)\ N\ \ : \nu X.T}\,\nu I$$

Figure 3–22: Typing for Mendler-style Corecursion

## 3.5 Syntax of Terms

Jackrabbit consists of a base language with conventional types, extended with the circle modality to express temporal properties, index operators that allow us to constrain types, and (co)recursive types for finite and infinite data structures.

Lower case letters such as $x$ are used for term variables. The unit type 1 has introduction form () and no elimination form. Function abstraction ($\lambda x.M$) is the introduction form and function application ($MN$) the elimination form for

$$\begin{array}{llll}
\text{Terms} & M, N & ::= & x \mid () \mid \lambda x.M \mid MN \mid \mathtt{fst}\ M \mid \mathtt{snd}\ M \mid (M, N) \\
 & & \mid & \mathtt{inl}\ M \mid \mathtt{inr}\ M \mid (\mathtt{case}\ \ M\ \ \mathtt{of}\ \ \mathtt{inl}\ \ a \mapsto N_0\ \mid \mathtt{inr}\ \ b \mapsto N_1) \\
 & & \mid & \mathtt{inj}\ M \mid \mathtt{rec}^{\Pi\overrightarrow{u{:}U}.(\mu Z.\Lambda\overrightarrow{u{:}U}.F)\vec{u}\to T}(\vec{u}; rf.M)\ \vec{C}\ N \\
 & & \mid & \mathtt{out}\ M \mid \mathtt{corec}^{\Pi\overrightarrow{u{:}U}.T\to(\nu Z.\Lambda\overrightarrow{u{:}U}.F)\vec{u}}(\vec{u}; rf.M)\ \vec{C}\ N \\
 & & \mid & \Lambda u.M \mid M\ C \mid \mathtt{pack}(C, M) \mid \mathtt{let}\ \mathtt{pack}(u, x) = M\ \mathtt{in}\ N \\
 & & \mid & \mathtt{refl} \mid \mathtt{eq}\ M_{C_1=C_2}\ \mathtt{with}\ (\Delta'; \rho' \mapsto N) \mid \mathtt{eqabort}^{A}_{C_1\neq C_2}\ M \\
 & & \mid & \bullet M \mid \mathtt{let}\ \ \bullet x = M\ \mathtt{in}\ N
\end{array}$$

Figure 3–23: Terms

function types →. Products (×) can package up two terms. They are introduced with $(M, N)$ and eliminated with the projections $\texttt{fst}\ M$ and $\texttt{snd}\ M$. Sums (+) are dual to products: whereas a pair gives us a single term $(M, N)$, composed of two terms $M$ and $N$, and eliminating can give us either component, a sum is created from either one of two terms $M$ or $N$ and eliminated by handling both cases. Sums are introduced with $\texttt{inl}\ M$ and $\texttt{inr}\ M$ and eliminated with $\texttt{case}\ M\ \texttt{of}\ \texttt{inl}\ a \mapsto N_0\ \mid \texttt{inr}\ b \mapsto N_1$. Recursive types are created with $\texttt{inj}\ M$ and eliminated with $\texttt{rec}^{\Pi u:U.(\mu Z.\Lambda\vec{u}.F)\vec{u}\to T}(\vec{u}; rf.M)\ \vec{C}\ N$. $\texttt{out}\ M$ is the elimination form for corecursive types and $\texttt{corec}^{\Pi u:U.T\to(\nu Z.\Lambda\vec{u}.F)\vec{u}}(\vec{u}; rf.M)\ \vec{C}\ N$ is the introduction form. For readability, we occasionally simplify the typing annotations for $\mu$ elimination and $\nu$ introduction, writing $A$ rather than $\Pi\overrightarrow{u{:}U}.(\mu Z.\Lambda\overrightarrow{u{:}U}.F)\vec{u} \to T$ or $\Pi\overrightarrow{u{:}U}.T \to (\nu Z.\Lambda\overrightarrow{u{:}U}.F)\vec{u}$.

$\Lambda u.M$ and $M\ \vec{C}$ are similar to $\lambda x.M$ and $MN$, but for indices rather than terms, so $u$ is an index variable and $\vec{C}$ an index term. $\texttt{pack}(C, M)$ is similar to $(M, N)$, but again with an index term $C$ instead of a term, and the second element can depend on the first. $\texttt{let}\ \texttt{pack}(u, x) = M\ \texttt{in}\ N$ is slightly different from the projections $\texttt{fst}\ M$ and $\texttt{snd}\ M$, as it uses both an index variable $u$ and a term variable $x$ that can appear in $N$. $\texttt{refl}$ witnesses index equality, $\texttt{eq}\ M_{C_1=C_2}\ \texttt{with}\ (\Delta'; \rho' \mapsto N)$ eliminates equal indices, and $\texttt{eqabort}^{A}_{C_1\neq C_2}\ M$ eliminates unequal indices. Finally, $\bullet M$ and $\texttt{let}\ \bullet x = M\ \texttt{in}\ N$ are the introduction and elimination forms for $\bigcirc$. We explain these rules in more depth in the section on typing.

$$\begin{aligned}
\texttt{vec}(C) &\equiv (\mu Z.\Lambda u{:}\mathbb{N}.\ u = \text{zero} \\
&+ (\Sigma v.u = \text{suc } v \times (\texttt{bool} \times Z\ v)))\ C \\
\texttt{nil} &\equiv \texttt{inj}\ (\texttt{inl}\ \texttt{refl}) \\
\texttt{cons}(\text{C}, hd, tl) &\equiv \texttt{inj}\ (\texttt{inr}\ (\texttt{pack}(\text{C}, (\texttt{refl}, (hd, tl)))))
\end{aligned}$$

Figure 3–24: Boolean Vector

### 3.5.1 Vectors

To give a taste of the syntax of the language and especially indices, we provide a few examples of vectors. A vector, defined in Figure 3–24, is a list that piggy-backs its length. Vectors can be used to enforce compile-time guarantees associated with length, which, avoiding runtime bounds checks or errors. We provide a few examples with boolean vectors in Jackrabbit.

**Example 4** (Vector Copy)**.** The copy function in Figure 3–25 takes an index $u$ and a vector of length $u$ and produces another vector with length $u$. The index and vector are passed to `rec` at the end, and these are substituted the body of the `rec` term, with $u$ for $u_1$ and $v$ for $r$. Because `rec` goes inside of the inductive object, `inj` has already been handled and we can case analyze the body of the inductive object. The `inl` case returns the empty vector and the `inr` case breaks the vector down into its constituents and builds it back up, passing the smaller $u_2$ to $f$ along with the tail of the vector.

**Example 5** (Vector Tail)**.** The function in Figure 3–26 takes a vector of length $u$ and a proof that $u$ is the successor of some number $u'$, and returns the vector's tail which has length $u$. On the type level, we make explicit the constraint $u = \text{suc } u'$ on the indices to produce a $\texttt{vec}(u')$ . The annotation for `rec`, however, uses a $\Pi$

```
$VecCpy : \Pi u{:}\mathbb{N}.\texttt{vec}(u) \to \texttt{vec}(u)$
$VecCpy \equiv \Lambda u.\lambda v.\texttt{rec}^{\Pi u:\mathbb{N}.\texttt{vec}(u)\to\texttt{vec}(u)}$
$(u_1; rf.$ case $r$ of
    inl $a \mapsto$ nil
  | inr $b \mapsto$ let pack$(u_2, x) = b$ in cons$(u_2,$ fst (snd $x), f\ u_2$ (snd (snd $x)))$
$)\ u\ v$
```

Figure 3–25: Vector Copy

```
$VecTail : \Pi u{:}\mathbb{N}.\Pi u'{:}\mathbb{N}.u = \text{suc } u' \to \texttt{vec}(u) \to \texttt{vec}(u')$
$VecTail \equiv \Lambda u.\Lambda u'.\lambda e.$
eq $e$ with $(u' : \mathbb{N}; \text{suc } u'/u \mapsto$
  $\lambda v.\texttt{rec}^{\Pi u:\mathbb{N}.\texttt{vec}(\text{suc } u)\to\texttt{vec}(u)}$
  $(u'; rf.$ case $r$ of
     inl $a \mapsto \texttt{eqabort}_{C_1\neq C_2}^{\texttt{vec}(u')}\ a$
  |  inr $b \mapsto$ let pack$(u_2, w) = b$ in snd (snd $w$))
   $u'\ v)$
```

Figure 3–26: Vector Tail

to place constraints on the input index. Since we cannot pattern match on indices in Jackrabbit, we rely on equality elimination to make the constraint implicit. This way, it can be used within the body of the recursor `rec`. For the `inl` case, we are given a proof that $u_1 = \text{zero}$. However, we already have a proof that $u = u_1 = \text{suc } u'$ so this is not unifiable. We therefore apply the $\texttt{eqabort}_{C_1\neq C_2}$ rule to get a term with the type we want, a vector whose length is the predecessor of zero. For the `inr` case, we simply return the tail without applying $f$. This is the same as applying the $p$ function from the section on recursion.

| | | |
|---|---|---|
| Regular substitution application | : | $[N/x]M$ |
| Future substitution application | : | $[N/x]^{\bullet}M$ |

Figure 3–27: Term Substitution

| | | |
|---|---|---|
| Type substitution $\tau$ | : | $A/Z$ |
| Type substitution application | : | $[\![\tau]\!]A \mid [\![\tau]\!]\Gamma$ |

Figure 3–28: Type Substitution

### 3.6 Substitution

The expression “let $x$ = 1 in 1 + $x$ = 3 - $x$” steps to “1 + 1 = 3 - 1”. Both of these expressions are closed, meaning that there are no free variables. When evaluating the first expression, we reach the open subexpression “1 + $x$ = 3 - $x$”, and need a mapping of variables to expressions so as to replace $x$ by 1. This is called a substitution.

In Jackrabbit, there are 6 kinds of substitutions. There are two kinds of substitutions at the type level: type substitutions for type variables in Figure 3–36, and index substitutions in Figure 3–37, which also appear in terms. In addition to index substitutions, which are shown in Figure 3–38, there are three other kinds of substitutions on terms. Regular substitutions are shown in Figure 3–33, future substitutions in Figure 3–34, for terms under a $\bullet$, and omega substitution for (co)recursive variables, since we make the recursive call with a function that should be agnostic to time. Omega substitution is defined almost exactly the same as regular substitution, and

| | | |
|---|---|---|
| Eternal substitution | : | $\tau;\theta$ |

$$\frac{}{\Delta \vdash \text{zero} : \mathbb{N}} \quad \frac{\Delta \vdash n : \mathbb{N}}{\Delta \vdash \text{suc}\ n : \mathbb{N}} \quad \frac{u : \mathbb{N} \in \Delta}{\Delta \vdash u : \mathbb{N}}$$

Figure 3–29: Peano Natural Numbers

we therefore do not show all of the cases. The exception is the case of the next-step substitution, shown in Figure 3–35.

### 3.6.1 Regular Substitution

Regular substitution, written $[N/x]M$ is displayed in Figure 3–27. It consists of replacing free instances of the term variable $x$ in term $M$ by term $N$. A free variable is a variable that is not bound. For example, $y$ is free in term $\lambda x.(x, y)$. A term with free variables is called an open term. We include side conditions to avoid variable capture. For example, $[N/x]\ \lambda x.x$ should produce the identity function, and $[x/y]\ \lambda x.y$ should not. Term variables are stored in the regular context $\Gamma$ along with their type.

An interesting case in Figure 3–33 is that of recursion. Although the substitution is applied to the inductive object $N$, it is not applied to the body $M$, since the only variables that appear free in $M$ are $f$, $r$, and $\vec{u}$. This also applies to corecursion, as well future and index substitutions.

### 3.6.2 Future Substitution

Future substitution, written $[N/x]^{\bullet}M$, is defined in Figure 3–34. Future substitutions replace term variables under a $\bullet$, which are kept with their type in the context $\Theta$. $[M'/x]^{\bullet}(\bullet M)$ is the most interesting case, and is defined as $\bullet([M'/x]M)$. Here, a future substitution becomes a regular substitution when stepping under a $\bullet$. This motivates the use of two contexts, since the objects in $\Gamma$ are replaced by those in $\Theta$ after stepping under the $\bullet$. For current variable substitutions, we have that

$$\begin{array}{lcl} \text{Index substitution } \theta & : & \cdot \mid \theta, C/u \\ \text{Index substitution application} & : & \{\theta\}M \mid \{\theta\}A \\ & \mid & \{\theta\}C \mid \{\theta\}\Gamma \mid \{\theta\}\theta' \end{array}$$

Figure 3–30: Index Substitution

$[N/x](\bullet M) \equiv \bullet M$ since the typing rules for $\bigcirc I$ ensure that $x$ cannot occur free in $M$.

Future substitutions don't require the same side conditions as those for regular substitutions. Variables are separated by ticks of the clock, so we can also separate variable namespaces. We define the operational semantics later, but as an example, the term $\texttt{let}\ \bullet x = \bullet 3\ \texttt{in}\ (x, \bullet x)$ steps to $(x, \bullet 3)$. The substitution is not applied to the first element of the pair.

### 3.6.3 Type Substitution

In a (co)recursive type, a type variable stands in for the (co)recursive type within its body. Type substitution retains type information when (co)iterating over a (co)inductive object. In a type substitution, we replace the type variable $Z$ by a type $A$. $A$ is a meta-variable that stands in for a type, whereas $Z$ is a type variable in Jackrabbit. Since type variables are also types, a type variable substitution can have the form $Z/Z$.

### 3.6.4 Index Substitution

Index substitution is shown in Figure 3–30. Indices can be substituted in terms, types, and other indices. Like the other substitutions, an index substitution is a mapping from variables to terms, in this case index terms. There are also some key differences. In Jackrabbit, index substitution uses simultaneous substitution rather than single substitution. Single substitutions are simpler, but simultaneous substitutions allow for a more general equality elimination rule. A single substitution maps a variable to a term, such as $2/x$ which maps $x$ to 2. A simultaneous substitution maps all variables to terms, and $\{\cdot, 2/x, 2/y\}(x + y)$ steps to $2 + 2$. We occasionally leave out $\cdot$, the empty substitution. Simultaneous substitutions can also be composed, but this is not necessary to substitute multiple variables. The substitution $u/u$ must also be added to $\sigma$ when substituting under a variable binding $u$, as simultaneous substitution is a total mapping.

Index substitution has a different motivation than that of the other substitutions. Regular and future substitutions make $\beta$-reduction possible for terms with variables such as case analysis and functions, and type substitution allows for well-behaved (co)recursion. Indices exist to place constraints on types, and so the key concept is that of unifiability. If two terms are unifiable, then there exists an instantiation of free variables such that the terms are equal. For example, consider the term "1 + $u$ = 3 - $u$". This comparison is true when we substitute 1 for $u$, and we say that the substitution $1/u$ unifies the terms "1 + $u$" and "3 - $u$". There can also be many unifiers for a particular term, as is the case for $u = v$, or none, as with $u \neq u + 1$.

For the term $u + 1 = v$, the substitution $2/u, 3/v$ acts as a unifier, along with $4/u, 5/v$ and an infinite number of other substitutions. We can unify these objects more generally with the substitution $u + 1/v$, producing $u + 1 = u + 1$. $u + 1/v$ is called a most general unifier of $u$ and $v$ ($\mathtt{mgu}(u, v)$). Formally, the substitution $\theta$ is a most general unifier of $u$ and $v$ whenever, for any other unifier $\theta'$ of $u$ and $v$, there is a substitution $\theta''$ such that the substitution $\{\theta''\}\theta$ is the same as $\theta'$. $u + 1 = u + 1$ is

$$\begin{array}{lcl} \{\theta\}\ \mathrm{zero} & \equiv & \mathrm{zero} \\ \{\theta\}\ \mathrm{suc}\ M & \equiv & \mathrm{suc}\ \{\theta\}M \\ \{\theta\}\ u & \equiv & C, \text{ where } C/u \in \theta \end{array}$$

Figure 3–31: Natural Number Substitution

still an open term. We can change this by applying a grounding substitution, which is substitution that produces a term with no free variables.

Although the domain for the index language is kept abstract, the examples given use an index language consisting of Peano natural numbers with variables. These are defined in Figure 3–29, and substitution on natural number indices is defined in Figure 3–31.

An index is well-formed according to the judgment $\Delta \vdash C : U$. This means that index $C$ from the index domain $U$ is well-formed in the context $\Delta$. We also make a few stipulations about the index language. Type-checking and unification must be decidable. That is, the unification algorithm will produce a unification if and only if two index objects are equal. We write $\Delta \vdash C_1 \neq C_2$ if $C_1$ is not equal to $C_2$. Index objects cannot depend on other index objects. We also place a few constraints on how substitution behaves [Jacob-Rao et al., 2018], which will be helpful in our substitution lemmas. $\Delta \vdash \theta : \Delta'$ means that for every $u_i$ in $\theta$ and $\Delta'$, $\Delta \vdash \{\theta\}u_i : \Delta'(u_i)$.

1. If $\Delta' \vdash \theta : \Delta$ and $\Delta \vdash C : U$ then $\Delta' \vdash C : \{\theta\}U$

2. If $\Delta_1 \vdash C : U$, $\Delta_2 \vdash \theta_1 : \Delta_1$, and $\Delta_3 \vdash \theta_2 : \Delta_2$ then $\Delta_3 \vdash \{\theta_2\}\{\theta_1\}C = \{\{\theta_2\}\theta_1\}C$

3. If $\Delta' \vdash \theta : \Delta$ and $\Delta \vdash C_1 = C_2$ then $\Delta' \vdash \{\theta\}C_1 = \{\theta\}C_2$

$$\frac{}{\Delta \vdash \cdot : \cdot} \qquad \frac{\Delta \vdash (\tau;\theta) : \Delta' \qquad \Delta \vdash C : U}{\Delta \vdash (\tau;\theta, C/u) : \Delta', u : U}$$

Figure 3–32: Well-typed Substitutions

The most interesting case for index substitution in Figure 3–38 is substitution in equality elimination: $\texttt{eq}\ M_{C_1=C_2}\ \texttt{with}\ (\Delta';\rho \mapsto N)$. If the elimination is a closed term, $M$ must be `refl`, and so any substitution must be unifying. However, this term is not necessarily closed and we have no guarantee that index variables will still unify under $\theta$. For this reason, applying a non-unifying substitution produces $\texttt{eqabort}^{A}_{C_1 \neq C_2}\ N$. If $\{\theta\}C_1$ and $\{\theta\}C_2$ do unify, then let $\rho'$ be the most general unifier, which exists in context $\Delta''$. Because $\rho$ is an `mgu` of $C_1$ and $C_2$, we know that any other unifier of $C_1$ and $C_2$ (in particular, $\{\rho'\}\theta$) can be written as the composition of some substitution $\theta'$ and of $\rho$. This substitution $\theta'$ can be applied to $N$, giving us the desired term.

### 3.7 Typing Rules

We describe the typing rules in five sections. Conventional operators have types $\rightarrow, \times, +$, and 1. Temporal operators give us the "reactive" part of Jackrabbit and act as a building block for the other temporal types presented earlier. Equality allows us to compare indices, and $\Pi$ and $\Sigma$ types let us quantify over index objects. Finally, the fixed point operators give us recursion and corecursion. This allows us to write programs that use finite data structures such as lists and trees and infinite data structures such as streams. The judgment $\Delta;\Theta;\Gamma \vdash M : A$ characterizes well-typed terms. Term $M$ has type $A$ under contexts $\Delta$, $\Theta$, and $\Gamma$, which are the contexts described in the previous section.

$$
\begin{array}{lcl}
[M'/x]\ x & \equiv & M' \\
[M'/x]\ y & \equiv & y, \text{ where } x \neq y \\
[M'/x]\ \lambda y.M & \equiv & \lambda y.[M'/x]M \\
 & & \text{where } x \neq y \text{ and } y \notin \texttt{FV}(M') \\
[M'/x]\ MN & \equiv & ([M'/x]M)([M'/x]N) \\
[M'/x]\ \texttt{fst}\ M & \equiv & \texttt{fst}\ [M'/x]M \\
[M'/x]\ \texttt{snd}\ M & \equiv & \texttt{snd}\ [M'/x]M \\
[M'/x]\ (M, N) & \equiv & ([M'/x]M, [M'/x]N) \\
[M'/x]\ \texttt{inl}\ M & \equiv & \texttt{inl}\ [M'/x]M \\
[M'/x]\ \texttt{inr}\ M & \equiv & \texttt{inr}\ [M'/x]M \\
[M'/x]\ (\texttt{case}\ \ M\ \ \texttt{of} & & \texttt{case}\ \ ([M'/x]M)\ \ \texttt{of} \\
\qquad\quad \texttt{| inl}\ \ a \mapsto N_0 & \equiv & \texttt{| inl}\ \ a \mapsto ([M'/x]N_0) \\
\qquad\quad \texttt{| inr}\ \ b \mapsto N_1) & & \texttt{| inr}\ \ b \mapsto ([M'/x]N_1) \\
 & & \text{where } a, b \notin \texttt{FV}(M') \\
[M'/x]\ \texttt{inj}\ M & \equiv & \texttt{inj}\ [M'/x]M \\
[M'/x]\ \texttt{out}\ M & \equiv & \texttt{out}\ [M'/x]M \\
[M'/x]\ \texttt{rec}^A(\vec{u}; rf.M)\ \vec{C}\ N & \equiv & \texttt{rec}^A(\vec{u}; rf.M)\ \vec{C}\ ([M'/x]N) \\
[M'/x]\ \texttt{corec}^A(\vec{u}; rf.M)\ \vec{C}\ N & \equiv & \texttt{corec}^A(\vec{u}; rf.M)\ \vec{C}\ ([M'/x]N) \\
[M'/x]\ \Lambda u.M & \equiv & \Lambda u.[M'/x]M \text{ where } u \notin \texttt{FV}(M') \\
[M'/x]\ M\ C & \equiv & ([M'/x]M)\ C \\
[M'/x]\ \texttt{pack}(C, M) & \equiv & \texttt{pack}(C, [M'/x]M) \\
[M'/x]\ \texttt{let}\,\texttt{pack}(u, y) = M & \equiv & \texttt{let}\,\texttt{pack}(u, y) = [M'/x]M \\
\qquad\quad \texttt{in}\ N & & \texttt{in}\ [M'/x]N \text{ where } y, u \notin \texttt{FV}(M') \\
[M'/x]\ \texttt{refl} & \equiv & \texttt{refl} \\
[M'/x]\ \texttt{eq}\ M_{C_1=C_2}\ \texttt{with}\ (\Delta; \theta \mapsto N) & \equiv & \texttt{eq}\ [M'/x]M_{C_1=C_2}\ \texttt{with}\ (\Delta; \theta \mapsto \\
 & & [\{\theta\}M'/x]N) \text{ where } \forall u_i \in \theta, u_i \notin \texttt{FV}(M') \\
[M'/x]\ \texttt{eqabort}^A_{C_1 \neq C_2}\ M & \equiv & \texttt{eqabort}^A_{C_1 \neq C_2}\ [M'/x]M \\
[M'/x]\ \bullet M & \equiv & \bullet M \\
[M'/x]\ \texttt{let}\ \bullet y = M\ \texttt{in}\ N & \equiv & \texttt{let}\ \bullet y = [M'/x]M\ \texttt{in}\ [M'/x]N \text{ where } y \notin \texttt{FV}(M')
\end{array}
$$

Figure 3–33: Substitutions on Terms

$$\begin{array}{lcl}
[M'/x]^{\bullet}\ y & \equiv & y, \text{ typing rules ensure that } x \neq y \\
[M'/x]^{\bullet}\ \bullet M & \equiv & \bullet\ [M'/x]M \\
[M'/x]^{\bullet}\ \texttt{let}\ \bullet y = M\ \texttt{in}\ N & \equiv & \texttt{let}\ \bullet y = [M'/x]^{\bullet}M\ \texttt{in}\ [M'/x]^{\bullet}N \\
[M'/x]^{\bullet}\ \lambda y.M & \equiv & \lambda y.[M'/x]^{\bullet}M \\
[M'/x]^{\bullet}\ MN & \equiv & ([M'/x]^{\bullet}M)([M'/x]^{\bullet}N) \\
[M'/x]^{\bullet}\ \texttt{fst}\ M & \equiv & \texttt{fst}\ [M'/x]^{\bullet}M \\
[M'/x]^{\bullet}\ \texttt{snd}\ M & \equiv & \texttt{snd}\ [M'/x]^{\bullet}M \\
[M'/x]^{\bullet}\ (M, N) & \equiv & ([M'/x]^{\bullet}M, [M'/x]^{\bullet}N) \\
[M'/x]^{\bullet}\ \texttt{inl}\ M & \equiv & \texttt{inl}\ [M'/x]^{\bullet}M \\
[M'/x]^{\bullet}\ \texttt{inr}\ M & \equiv & \texttt{inr}\ [M'/x]^{\bullet}M \\
[M'/x]^{\bullet}\ (\texttt{case}\ \ M\ \ \texttt{of} & & \texttt{case}\ \ ([M'/x]^{\bullet}M)\ \ \texttt{of} \\
\qquad\quad \texttt{|}\ \texttt{inl}\ \ a \mapsto N_0 & \equiv & \texttt{|}\ \texttt{inl}\ \ a \mapsto ([M'/x]^{\bullet}N_0) \\
\qquad\quad \texttt{|}\ \texttt{inr}\ \ b \mapsto N_1) & & \texttt{|}\ \texttt{inr}\ \ b \mapsto ([M'/x]^{\bullet}N_1) \\
[M'/x]^{\bullet}\ \texttt{inj}\ M & \equiv & \texttt{inj}\ [M'/x]^{\bullet}M \\
[M'/x]^{\bullet}\ \texttt{out}\ M & \equiv & \texttt{out}\ [M'/x]^{\bullet}M \\
[M'/x]^{\bullet}\ \texttt{rec}^{A}(\vec{u}; rf.M)\ \vec{C}\ N & \equiv & \texttt{rec}^{A}(\vec{u}; rf.M)\ \vec{C}\ ([M'/x]^{\bullet}N) \\
[M'/x]^{\bullet}\ \texttt{corec}^{A}(\vec{u}; rf.M)\ \vec{C}\ N & \equiv & \texttt{corec}^{A}(\vec{u}; rf.M)\ \vec{C}\ ([M'/x]^{\bullet}N) \\
[M'/x]^{\bullet}\ \Lambda u.M & \equiv & \Lambda u.[M'/x]^{\bullet}M \\
[M'/x]^{\bullet}\ M\ C & \equiv & ([M'/x]^{\bullet}M)\ C \\
[M'/x]^{\bullet}\ \texttt{pack}(C, M) & \equiv & \texttt{pack}(C, [M'/x]^{\bullet}M) \\
[M'/x]^{\bullet}\ \texttt{let}\ \texttt{pack}(u, y) = M & \equiv & \texttt{let}\ \texttt{pack}(u, y) = [M'/x]^{\bullet}M \\
\qquad\quad \texttt{in}\ N & & \texttt{in}\ [M'/x]^{\bullet}N \\
[M'/x]^{\bullet}\ \texttt{refl} & \equiv & \texttt{refl} \\
[M'/x]^{\bullet}\ \texttt{eq}\ M_{C_1=C_2}\ \texttt{with}\ (\Delta; \theta \mapsto N) & \equiv & \texttt{eq}\ [M'/x]^{\bullet}M_{C_1=C_2}\ \texttt{with}\ (\Delta; \theta \mapsto \\
 & & [\{\theta\}M'/x]^{\bullet}N) \text{ where } \forall u_i \in \theta, u_i \notin \texttt{FV}(M') \\
[M'/x]^{\bullet}\ \texttt{eqabort}^{A}_{C_1 \neq C_2}\ M & \equiv & \texttt{eqabort}^{A}_{C_1 \neq C_2}\ [M'/x]^{\bullet}M
\end{array}$$

Figure 3–34: Future Substitutions

$$[M'/x]^{\omega}\ \ \bullet M \ \ \equiv\ \ \bullet\ [M'/x]^{\omega}M$$

Figure 3–35: Omega Substitution

$$\begin{array}{lcl}
\llbracket\tau\rrbracket\ Z & \equiv & A, \text{ where } A/Z \in \tau \\
\llbracket\tau\rrbracket\ \mu Z.\Lambda\overrightarrow{u{:}U}.A & \equiv & \mu Z.\Lambda\overrightarrow{u{:}U}.\llbracket\tau, Z/Z\rrbracket A, \text{ where } Z \notin \mathtt{FV}(\tau) \\
\llbracket\tau\rrbracket\ \nu Z.\Lambda\overrightarrow{u{:}U}.A & \equiv & \nu Z.\Lambda\overrightarrow{u{:}U}.\llbracket\tau, Z/Z\rrbracket A, \text{ where } Z \notin \mathtt{FV}(\tau) \\
\llbracket\tau\rrbracket\ 1 & \equiv & 1 \\
\llbracket\tau\rrbracket\ A \to B & \equiv & \llbracket\tau\rrbracket A \to \llbracket\tau\rrbracket B \\
\llbracket\tau\rrbracket\ A \times B & \equiv & \llbracket\tau\rrbracket A \times \llbracket\tau\rrbracket B \\
\llbracket\tau\rrbracket\ A + B & \equiv & \llbracket\tau\rrbracket A + \llbracket\tau\rrbracket B \\
\llbracket\tau\rrbracket\ \bigcirc A & \equiv & \bigcirc(\llbracket\tau\rrbracket A) \\
\llbracket\tau\rrbracket\ R\ \vec{C} & \equiv & (\llbracket\tau\rrbracket R)\ \vec{C} \\
\llbracket\tau\rrbracket\ \Pi u{:}U.A & \equiv & \Pi u{:}U.\llbracket\tau\rrbracket A \\
\llbracket\tau\rrbracket\ \Sigma u{:}U.A & \equiv & \Sigma u{:}U.\llbracket\tau\rrbracket A \\
\llbracket\tau\rrbracket\ C_1 = C_2 & \equiv & C_1 = C_2
\end{array}$$

Figure 3–36: Type Substitutions

$$\begin{array}{lcl}
\{\theta\}\ Z & \equiv & Z \\
\{\theta\}\ \mu Z.\Lambda\overrightarrow{u{:}U}.R & \equiv & \mu Z.\Lambda\overrightarrow{u{:}U}.A \\
\{\theta\}\ \nu Z.\Lambda\overrightarrow{u{:}U}.R & \equiv & \nu Z.\Lambda\overrightarrow{u{:}U}.A \\
\{\theta\}\ 1 & \equiv & 1 \\
\{\theta\}\ A \to B & \equiv & \{\theta\}A \to \{\theta\}B \\
\{\theta\}\ A \times B & \equiv & \{\theta\}A \times \{\theta\}B \\
\{\theta\}\ A + B & \equiv & \{\theta\}A + \{\theta\}B \\
\{\theta\}\ \bigcirc A & \equiv & \bigcirc(\{\theta\}A) \\
\{\theta\}\ R\ \vec{C} & \equiv & (\{\theta\}R)\ (\{\theta\}\vec{C}) \\
\{\theta\}\ \Pi u{:}U.A & \equiv & \Pi u{:}(\{\theta\}U).\{\theta, u/u\}A, \text{ where } u \notin \mathtt{FV}(\theta) \\
\{\theta\}\ \Sigma u{:}U.A & \equiv & \Sigma u{:}(\{\theta\}U).\{\theta, u/u\}A, \text{ where } u \notin \mathtt{FV}(\theta) \\
\{\theta\}\ C_1 = C_2 & \equiv & \{\theta\}C_1 = \{\theta\}C_2
\end{array}$$

Figure 3–37: Index Substitutions on Types

$$\begin{array}{lcl}
\{\theta\}\ x & \equiv & x \\
\{\theta\}\ \lambda x.M & \equiv & \lambda x.\{\theta\}M \\
\{\theta\}\ M N & \equiv & (\{\theta\}M)(\{\theta\}N) \\
\{\theta\}\ \texttt{fst}\ M & \equiv & \texttt{fst}\ \{\theta\}M \\
\{\theta\}\ \texttt{snd}\ M & \equiv & \texttt{snd}\ \{\theta\}M \\
\{\theta\}\ (M, N) & \equiv & (\{\theta\}M, \{\theta\}N) \\
\{\theta\}\ \texttt{inl}\ M & \equiv & \texttt{inl}\ \{\theta\}M \\
\{\theta\}\ \texttt{inr}\ M & \equiv & \texttt{inr}\ \{\theta\}M \\
\{\theta\}\ (\texttt{case}\ \ M\ \ \texttt{of} & & \texttt{case}\ \ (\{\theta\}M)\ \ \texttt{of} \\
\quad\ \ \texttt{|}\ \texttt{inl}\ \ a \mapsto N_0 & \equiv & \texttt{|}\ \texttt{inl}\ \ a \mapsto (\{\theta\}N_0) \\
\quad\ \ \texttt{|}\ \texttt{inr}\ \ b \mapsto N_1) & & \texttt{|}\ \texttt{inr}\ \ b \mapsto (\{\theta\}N_1) \\
\{\theta\}\ \texttt{inj}\ M & \equiv & \texttt{inj}\ \{\theta\}M \\
\{\theta\}\ \texttt{out}\ M & \equiv & \texttt{out}\ \{\theta\}M \\
\{\theta\}\ \texttt{rec}^{A}(\vec{u}; rf.M)\ \vec{C}\ N & \equiv & \texttt{rec}^{A}(\vec{u}; rf.M)\ (\{\theta\}\vec{C})\ (\{\theta\}N) \\
\{\theta\}\ \texttt{corec}^{A}(\vec{u}; rf.M)\ \vec{C}\ N & \equiv & \texttt{corec}^{A}(\vec{u}; rf.M)\ (\{\theta\}\vec{C})\ (\{\theta\}N) \\
\{\theta\}\ \Lambda u.M & \equiv & \Lambda u.\{\theta, u/u\}M, \text{ where } u \notin \texttt{FV}(\theta) \\
\{\theta\}\ M\ C & \equiv & (\{\theta\}M)\ (\{\theta\}C) \\
\{\theta\}\ \texttt{pack}(C, M) & \equiv & \texttt{pack}(\{\theta\}C, \{\theta\}M) \\
\{\theta\}\ \texttt{let}\,\texttt{pack}(u, x) = M & \equiv & \texttt{let}\,\texttt{pack}(u, x) = \{\theta\}M \\
\quad\ \ \texttt{in}\ N & & \texttt{in}\ \{\theta, u/u\}N, \text{ where } u \notin \texttt{FV}(\theta) \\
\{\theta\}\ \texttt{refl} & \equiv & \texttt{refl} \\
\{\theta\}\ \texttt{eq}\ M_{C_1 = C_2}\ \texttt{with}\ (\Delta'; \rho \mapsto N) & \equiv & \texttt{eq}\ \{\theta\}M_{C_1 = C_2}\ \texttt{with}\ (\Delta''; \rho' \mapsto \{\theta'\}N), \\
 & & \text{for } \theta' \text{ s.t. } \Delta'' \vdash \theta' : \Delta' \text{ and } \{\rho'\}\theta = \{\theta'\}\rho, \\
 & & \text{where } \Delta \vdash \texttt{unify}(\{\theta\}C_1, \{\theta\}C_2) \Rightarrow (\rho'; \Delta'') \\
\{\theta\}\ \texttt{eq}\ M_{C_1 = C_2}\ \texttt{with}\ (\Delta'; \rho \mapsto N) & \equiv & \texttt{eqabort}^{A}_{C_1 \neq C_2}\ \{\theta\}\ M, \\
 & & \text{where } \Delta \vdash \texttt{unify}(\{\theta\}C_1, \{\theta\}C_2) \Rightarrow \bot \\
 & & \text{and N} : A \\
\{\theta\}\ \texttt{eqabort}^{A}_{C_1 \neq C_2}\ M & \equiv & \texttt{eqabort}^{A}_{C_1 \neq C_2}\ \{\theta\}M \\
\{\theta\}\ \bullet M & \equiv & \bullet\{\theta\}M \\
\{\theta\}\ \texttt{let}\ \bullet x = M\ \texttt{in}\ N & \equiv & \texttt{let}\ \bullet x = \{\theta\}M\ \texttt{in}\ \{\theta\}N
\end{array}$$

Figure 3–38: Index Substitutions on Terms

$$\frac{x{:}A \in \Gamma}{\Delta; \Theta; \Gamma \vdash x : A}\, xI \qquad \frac{}{\Delta; \Theta; \Gamma \vdash () : 1}\, 1I \qquad \frac{\Delta; \Theta; \Gamma, x : A \vdash M : B}{\Delta; \Theta; \Gamma \vdash \lambda x.M : A \to B} \to I$$

$$\frac{\Delta; \Theta; \Gamma \vdash M : A \times B}{\Delta; \Theta; \Gamma \vdash \texttt{fst}\ M : A} \times E_1 \qquad \frac{\Delta; \Theta; \Gamma \vdash M : A \to B \qquad \Delta; \Theta; \Gamma \vdash N : A}{\Delta; \Theta; \Gamma \vdash MN : B} \to E$$

$$\frac{\Delta; \Theta; \Gamma \vdash M : A \times B}{\Delta; \Theta; \Gamma \vdash \texttt{snd}\ M : B} \times E_2 \qquad \frac{\Delta; \Theta; \Gamma \vdash M : A \qquad \Delta; \Theta; \Gamma \vdash N : B}{\Delta; \Theta; \Gamma \vdash (M, N) : A \times B} \times I$$

$$\frac{\Delta; \Theta; \Gamma \vdash M : A}{\Delta; \Theta; \Gamma \vdash \texttt{inl}\ M : A + B} + I_1 \qquad \frac{\Delta; \Theta; \Gamma \vdash M : B}{\Delta; \Theta; \Gamma \vdash \texttt{inr}\ M : A + B} + I_2$$

$$\frac{\Delta; \Theta; \Gamma \vdash M : A + B \qquad \Delta; \Theta; \Gamma, x : A \vdash N_1 : T \qquad \Delta; \Theta; \Gamma, y : B \vdash N_2 : T}{\Delta; \Theta; \Gamma \vdash \texttt{case}\ M\ \texttt{of}\ \texttt{inl}\ x \mapsto N_1\ \mid \texttt{inr}\ y \mapsto N_2 : T} + E$$

Figure 3–39: Conventional Types

$$\frac{\begin{array}{r} \Delta \vdash \texttt{unify}(C_1, C_2) \Rightarrow (\theta; \Delta') \\ \Delta; \Theta; \Gamma \vdash M : C_1 = C_2 \qquad \Delta'; \{\theta\}\Theta; \{\theta\}\Gamma \vdash N : \{\theta\}A \end{array}}{\Delta; \Theta; \Gamma \vdash \texttt{eq}\ M_{C_1 = C_2}\ \texttt{with}\ (\Delta'; \theta \mapsto N) : A} = E_1$$

$$\frac{\Delta; \Theta; \Gamma \vdash M : C_1 = C_2 \qquad \Delta \vdash \texttt{unify}(C_1, C_2) \Rightarrow \bot}{\Delta; \Theta; \Gamma \vdash \texttt{eqabort}^{A}_{C_1 \neq C_2}\ M : A} = E_2$$

$$\frac{\Delta \vdash C : U}{\Delta; \Theta; \Gamma \vdash \texttt{refl} : C = C} = I$$

Figure 3–40: Equality Types

### 3.7.1 Conventional Operators

Functions (→) have a single introduction and elimination form, products (×) have two elimination forms (taking either element of the pair), and sums (+), being dual to products, have two introduction forms (wrapping an element of either type).

### 3.7.2 Equality

The equality introduction rule `refl` witnesses the equality of an index object with itself. There are two ways to use an equality type. Using $=E_2$, we can derive any type we want from the contradiction that two index objects are equal but not unifiable.

The other rule, $=E_1$, is a bit more involved. If two indices are equal and we can derive a type $A$ which relies on a most general unifier $\theta$, then because $\theta$ is a most general unifier, $A$ is derivable under any substitution. The $=E_1$ rule makes it possible to derive $A$ without $\theta$ [Schroeder-Heister, 1993]. This rule is a variation of the rule in Figure 3–41, written without proof terms for simplicity. In the latter, index terms can be substituted in either direction, whereas $=E_1$ forces unification. For example, if $\Delta \vdash u = \text{suc } v$, then $P$ (suc $v$) can be derived from $P\ u$ and vice-versa. With $=E_1$, there is only one `mgu`, suc $v/u$. $P$ (suc $v$) is therefore not derivable from $P\ u$.

We also note that the premise $M$ is not necessary, since we also have the premise $\texttt{unify}(C_1, C_2) \Rightarrow (\theta; \Delta')$. However, the type $C_1 = C_2$ does need to be added to a context in the conclusion of the rule. Placing $M$ in the premise abstracts the details of which context to add it to. This also simplifies the proof of type preservation, since an equality term only steps when $M = \texttt{refl}$.

To prove type preservation, we use an index substitution lemma. Proving substitution lemmas with equality elimination presents a problem, since we lose information about whether a given substitution is a unifier. In an open context, $\{\theta\}C_1 = \{\theta\}C_2$ is not derivable from $C_1 = C_2$. There are a few approaches to handle this. Schürmann and Pfenning [1998] resolve this using environments, since an environment provides the entire history of a unification. Baelde and Nadathur [2012] introduce a rule which incorporates any substitutions that are applied, so that the history of the unification is provided as part of the term. Our approach is to step to $\texttt{eqabort}^{A}_{C_1 \neq C_2}\ M$ under

$$\frac{\Delta;\Theta;\Gamma \vdash C_1 = C_2 \qquad \Delta;\Theta;\Gamma \vdash A\ C_1}{\Delta;\Theta;\Gamma \vdash A\ C_2}$$

Figure 3–41: Simple Equality Elimination

$$\frac{\Delta;\Theta;\Gamma \vdash M : \{C/u\}A \qquad \Delta \vdash C : U}{\Delta;\Theta;\Gamma \vdash \texttt{pack}(C, M) : \Sigma u{:}U.A}\ \Sigma I$$

$$\frac{\Delta;\Theta;\Gamma \vdash M : \Sigma u{:}U.A \qquad \Delta, u{:}U;\Theta;\Gamma, x{:}A \vdash N : B}{\Delta;\Theta;\Gamma \vdash \texttt{let pack}(u, x) = M \ \texttt{in}\ N : B}\ \Sigma E$$

$$\frac{\Delta, u{:}U;\Theta;\Gamma \vdash M : A}{\Delta;\Theta;\Gamma \vdash \Lambda u.M : \Pi u{:}U.A}\ \Pi I \qquad \frac{\Delta;\Theta;\Gamma \vdash M : \Pi u{:}U.A \qquad \Delta \vdash C : U}{\Delta;\{C/u\}\Theta;\{C/u\}\Gamma \vdash M\ C : \{C/u\}A}\ \Pi E$$

Figure 3–42: Quantification Types

a non-unifying substitution. This is a term with an arbitrary type, but a program's type is still preserved under evaluation. Consider the vector tail function, where the `inl` case has type $u =$ zero. It is not possible to step to this, since any time that $M =$ `refl` will be an `inr` case. However, for the program to typecheck, we have to address the `inl` case. Once we step into the body of $= E_1$, we find that $u$ has been substituted with suc $u'$. $M$ gives us the guarantee that $C_1 = C_2$ and so, given the case that suc $u' = z$, we are justified in deriving whatever type we wish.

### 3.7.3 Quantification

Recall that Π-kinded types correspond to predicates in first-order logic, and that indices correspond to objects. A pack is also known as a dependent pair, since it consists of two objects, where the second can depend on the first. This corresponds to a proof of an existential claim. To prove a universal claim, the burden is to show that a property holds for *any* object in some domain. This is done with $\Lambda u.M$ since

$$\frac{\Delta;\Theta;\Gamma \vdash M : \llbracket \mu Z{:}K.\Lambda \overrightarrow{u{:}U}.S/Z \rrbracket \{\vec{C}/\vec{u}\} S}{\Delta;\Theta;\Gamma \vdash \texttt{inj}\ M : (\mu Z{:}K.\Lambda \overrightarrow{u{:}U}.S)\vec{C}}\ \mu I$$

$$\frac{\begin{array}{c}\mathcal{F} \equiv \Pi \vec{w}{:}U.Z\vec{w} \to \{\vec{w}/\vec{u}\}T \\ (\Phi, Z{:}K; \overrightarrow{u{:}U}; f{:}\mathcal{F}); \cdot; r{:}S \vdash M{:}T \qquad (\Phi;\Psi);\Theta;\Gamma \vdash N{:}(\mu Z{:}K.\Lambda \overrightarrow{u{:}U}.S)\vec{C}\end{array}}{(\Phi;\Psi);\Theta;\Gamma \vdash \texttt{rec}^{\Pi \overrightarrow{u{:}U}.(\mu Z{:}K.\Lambda \vec{u}.S)\vec{u} \to T}(\vec{u}; r f.M)\ \vec{C}\ N : \{\vec{C}/\vec{u}\}T}\ \mu E$$

$$\frac{\Delta;\Theta;\Gamma \vdash M : (\nu Z{:}K.\Lambda \overrightarrow{u{:}U}.S)\vec{C}}{\Delta;\Theta;\Gamma \vdash \texttt{out}\ M : \llbracket \nu Z{:}K.\Lambda \overrightarrow{u{:}U}.S/Z \rrbracket \{\vec{C}/\vec{u}\} S}\ \nu E$$

$$\frac{\begin{array}{c}\mathcal{F} \equiv \Pi \vec{w}{:}U.\{\vec{w}/\vec{u}\}T \to Z\vec{w} \\ (\Phi, Z{:}K; \overrightarrow{u{:}U}; f{:}\mathcal{F}); \cdot; r{:}T \vdash M{:}S \qquad (\Phi;\Psi);\Theta;\Gamma \vdash N{:}\{\vec{C}/\vec{u}\}T\end{array}}{(\Phi;\Psi);\Theta;\Gamma \vdash \texttt{corec}^{\Pi \overrightarrow{u{:}U}.T \to (\nu Z{:}K.\Lambda \vec{u}.S)\vec{u}}(\vec{u}; r f.M)\ \vec{C}\ N : (\nu Z{:}K.\Lambda \overrightarrow{u{:}U}.S)\vec{C}}\ \nu I$$

Figure 3–43: (Co)Recursive Types

the proof $M$ must hold for any object substituted for $u$. This is also known as a dependent function, since it can be thought of as a function of type $U \to A$.

Unlike pairs in Jackrabbit, which can be eliminated in two ways, there is only one elimination rule for $\Sigma$. The $\Sigma E$ matches the first element of a dependent pair $M$ with the variable $u$, and the second with $x$. These variables can then appear in any body $N$. $M$ has $\Sigma$ type and will evaluate to some $\texttt{pack}(C, M')$.

### 3.7.4 (Co)Recursion

In the recursive and corecursive rules, variables $f$, $r$, and $\vec{u}$ are accessible in the body. The object over which the recursion is done is $r$, $f$ is a function, and $\vec{u}$ are the indices.

Mendler-style recursion [Mendler, 1988] guarantees termination by constraining recursive calls. The only calls that can be done to an object of type $Z\ \vec{C}$ is to either leave the variable as is or to apply the provided functions $f$ (after applying it to an index). What's more, we don't have access to the entire inductive object: we only have $r$, in which the outer layer is peeled off, giving it type $S$ rather than $\mu Z{:}K.\Lambda\overrightarrow{u{:}U}.S$.

In the original presentation, Mendler recursion uses combinators. At each recursive step, new types for are passed as arguments to a polymorphic combinator to replace type variables in the inductive and output types. In Jackrabbit, $\Pi$ types only work with indices, so the type mapping is taken directly from the eternal context. This is similar to Abel [2004], which also uses a subtyping relation $X \leq \mu X.F$.

Every time we step into the body of a (co)recursive object, we clear the index contexts and the term contexts $\Gamma$ and $\Theta$, and introduce index variables $\overrightarrow{u{:}U}$ and the term variables $f$ and $r$. Although it is called the eternal context, index variables not persist through (co)recursion. The eternal aspect refers to the behaviour under the next step modality, where normal variables are cleared at each step. It is simple to pass indices forward though. The type variable context is not cleared, however, which makes it possible to create a fair interleaving of two types. An interleaving type has the form $\mu X.\mu Y.A$, where both $X$ and $Y$ are free in $A$. For example, the fairness property $A$ infinitely often ($\Box\Diamond A$) can be expressed as $\mu X.\mu Y.(A \times \bigcirc X) + \bigcirc Y$.

The corecursive rules allow for programs that deal with infinite data structures. Corecursion is dual to recursion, so whereas the recursive rule is the $\mu$ elimination rule, the corecursive rule is the $\nu$ introduction rule. In the rec rule, $f : \Pi\vec{w}{:}U.Z\vec{w} \rightarrow \{\vec{w}/\vec{u}\}T$. For corec, $f : \Pi\vec{w}{:}U.\{\vec{w}/\vec{u}\}T \rightarrow Z\vec{w}$. This duality can also be seen in the `inj` and `out` rules: the `inj` rule folds up an inductive type and the `out`

$$\frac{\Delta; \cdot; \Theta \vdash M : A}{\Delta; \Theta; \Gamma \vdash \bullet M : \bigcirc A}\bigcirc I \qquad \frac{\Delta; \Theta; \Gamma \vdash M : \bigcirc A \qquad \Delta; \Theta, x : A; \Gamma \vdash N : C}{\Delta; \Theta; \Gamma \vdash \texttt{let } \bullet x = M \texttt{ in } N : C}\bigcirc E$$

Figure 3–44: Temporal Types

rule unfolds a coinductive type. For example, if `inl` $a : A + \bigcirc(\mu Z.A + \bigcirc Z)$, then `inj inl` $a : \mu Z.A + \bigcirc Z$. In the corecursive case, if $as : \nu Z.A \times \bigcirc Z$ then `out` $as : A \times \bigcirc(\nu Z.A \times \bigcirc Z)$.

The recursive and corecursive rules require their arguments to be applied. To write a function that takes an arbitrary (co)inductive object, we therefore write $\Lambda\vec{w}.\lambda y.\texttt{rec}^{\Pi\vec{u}.(\mu Z:K.\Lambda\vec{u}.S)\vec{u}\to T}(\vec{u}; rf.M)\;\vec{w}\;y$, rather than $\texttt{rec}^{\Pi\vec{u}.(\mu Z:K.\Lambda\vec{u}.S)\vec{u}\to T}(\vec{u}; rf.M)$ .

### 3.7.5 Temporal Typing Rules

The next-step modality gives us the reactive part of Jackrabbit and is the basis for the other temporal types $\Diamond$, $\Box$, and $\mathcal{U}$. When we step under a circle-modality, $\Gamma$ is flushed and objects from $\Theta$ are transferred to the $\Gamma$ context. For an object to exist in the future, it cannot contain any variables from the current time. This prevents space leaks [Krishnaswami, 2013]. To use a variable under a $\bullet$, it has to be in $\Theta$ before going under the $\bullet$, and for this to happen it must be introduced with the $\bigcirc E$ rule. So to use a variable in the future, it must be from the future.

### 3.7.6 Values

A value is a term that has been fully evaluated. In our setting, whether a term can be fully evaluated is a function of time. We define $\alpha$-values constructively, in contrast to Cave et al. [2014], who define values as complementary to unevaluated terms. Figure 3–45 shows 0-values, also written `val` $v$. These are terms that are fully evaluated at the first time-step. $\alpha$-values are defined with `val` $v$ in conjunction

$$\begin{array}{lll} v & ::= & () \mid \lambda x.M \mid (v_1, v_2) \mid \texttt{inl}\ v \mid \texttt{inr}\ v \mid \texttt{inj}\ v \mid \Lambda u.M \mid \texttt{pack}(C, v) \\ & \mid & \texttt{refl} \mid \texttt{eqabort}^{A}_{C_1 \neq C_2}\ M \mid \texttt{corec}^{T}(\vec{u}; rf.M)\ \vec{C}\ N \end{array}$$

Figure 3–45: 0-values

$$\frac{\texttt{val}_n\ v}{\texttt{val}_{n+1}\ \bullet v}$$

Figure 3–46: $\alpha$-values

with the rule in Figure 3–46. A term that is fully evaluated at some time $\alpha \leq \omega$ is called an $\omega$-value.

Since $\texttt{corec}^{T}(\vec{u}; rf.M)\ \vec{C}\ N$ is an infinite structure, it cannot be evaluated in its entirety. We consider $\texttt{corec}^{T}(\vec{u}; rf.M)\ \vec{C}\ N$ to be a value for the proof of progress, but we note that it behaves differently from other values, since it can generate values. Just as functions are treated as first class and can be passed around, so can corecursive objects. A corecursive object is an $\alpha$-value, meaning that given enough clock ticks (in particular, $\alpha$), it will be considered a value. A function can only receive its argument once, however. For some corecursive objects, it is always possible to produce a value with `out`. Infinite evaluation is prevented by the operational semantics, however. Similar to a thunk, evaluation of $\texttt{corec}^{T}(\vec{u}; rf.M)\ \vec{C}\ N$ is prevented until called upon with `out`.

### 3.8 Operational Semantics

The operational semantics of a language is how a term steps to a value. In Jackrabbit, there are two kinds of stepping rules. The first, $\longrightarrow$, is small-step $\beta$-reduction. The other stepping rule, $\leadsto_\alpha$, is indexed by a number $\alpha$, which represents the number of clock ticks into the future that a term can step.

### 3.8.1 Reduction Rules

The use of a small-step operational semantics in Jackrabbit follows Cave et al. [2014]. This is in contrast to a big-step semantics, whose rules incorporate reduction to values. The congruence rules in Jackrabbit in the next section also incorporate values, but these are to ensure that reduction is deterministic. Reduction rules for pairs, functions, and sums, shown in Figure 3–47, are straightforward. Dependent function application for $\Pi$ and let-elimination for $\bigcirc$ are similar, but the former steps to an index substitution and the latter a $\bullet$ substitution. Elimination for $\Sigma$ uses two substitutions – one index and one regular, and we recall that the type of $M$ can depend on $C$ as this is important in the order of our substitutions. For the equality elimination rule $=E_1$, we can step to $N$ whenever we know that the two indices are equal ($M = \texttt{refl}$), and that $\theta$ is a most general unifier of the indices in the context $\Delta$.

Lastly are the recursive and corecursive rules. Each of them steps to three substitutions in the body of the (co)recursive object. The function $f$ is the recursive call. Since it's a recursive call, we replace $f$ applied to its arguments (the indices followed by the inductive object) with another $\mu$ elimination term (or $\nu$ introduction in the corecursive case). Since $f$ lives in the eternal context, we use omega substitution. This allows us to make the recursive call under a $\bullet$. We cannot use $f$ to save objects for later, however, since the function only serves to apply indices and to indicate where to apply type substitution. We also annotate the term $M$ with a type substitution $[\![A/Z]\!]$. The type variable is safely be replaced by the recursive type here. If this were done earlier, it would be possible to make the recursive call on an

$$
\begin{array}{lll}
(\lambda x.M)N & \longrightarrow & [N/x]M \\
\texttt{fst}\ (M, N) & \longrightarrow & M \\
\texttt{snd}\ (M, N) & \longrightarrow & N \\
\texttt{case}\ (\texttt{inl}\ M)\ \texttt{of}\ \texttt{inl}\ x \mapsto N_1\ \mid \texttt{inr}\ y \mapsto N_2 & \longrightarrow & [M/x]N_1 \\
\texttt{case}\ (\texttt{inr}\ M)\ \texttt{of}\ \texttt{inl}\ x \mapsto N_1\ \mid \texttt{inr}\ y \mapsto N_2 & \longrightarrow & [M/y]N_2 \\
\texttt{let}\ \bullet x = \bullet M\ \texttt{in}\ N & \longrightarrow & [M/x]^{\bullet}N \\
(\Lambda u.M)C & \longrightarrow & \{C/u\}M \\
\texttt{let}\,\texttt{pack}(u, x) = \texttt{pack}(C, M)\ \texttt{in}\ N & \longrightarrow & [M/x]\{C/u\}N \\
\texttt{eq}\,\texttt{refl}_{C_1=C_2}\ \texttt{with}\ (\Delta'; \theta \mapsto N) & \longrightarrow & N\text{, where} \\
 & & \Delta \vdash \texttt{unify}(C_1, C_2) \Rightarrow (\theta; \Delta') \\
\texttt{rec}^{A}(\vec{u}; rf.M)\ \vec{C}\ (\texttt{inj}\ N) & \longrightarrow & \\
\multicolumn{3}{l}{[N/r][\Lambda\vec{w}.\lambda y.(\texttt{rec}^{A}(\vec{u}; rf.M)\ \vec{w}\ y)/f]^{\omega}\{\vec{C}/\vec{u}\}M^{\llbracket A/Z \rrbracket}} \\
\texttt{out}\ (\texttt{corec}^{A}(\vec{u}; rf.M)\ \vec{C}\ N) & \longrightarrow & \\
\multicolumn{3}{l}{[N/r][\Lambda\vec{w}.\lambda y.(\texttt{corec}^{A}(\vec{u}; rf.M)\ \vec{w}\ y)/f]^{\omega}\{\vec{C}/\vec{u}\}M^{\llbracket A/Z \rrbracket}}
\end{array}
$$

Figure 3–47: Reduction Rules

object that is not strictly smaller. We also replace the variable $r$ with the inductive object, as well as the index variables $\vec{u}$ with indices $\vec{C}$. Since $\vec{C}$ can already occur in $N$, we apply the index substitution first.

The main difference between the recursive and corecursive reduction rules are the placement of `inj` and `out`. The input of `rec` is an inductive object, and we peel off the layers as we recurse. `corec`, on the other hand, returns a corecursive object which can be eliminated with `out` .

### 3.8.2 Evaluation Contexts

Evaluation contexts are often used as syntactic sugar for congruence rules to simplify the presentation of simpler stepping rules. An evaluation context hole specifies where reduction can occur. For example, $\mathcal{E}N$ corresponds to the congruence

$$\frac{M \longrightarrow M'}{M\ N \longrightarrow M'\ N}$$

Figure 3–48: $MN$ Stepping Rule

$$\begin{array}{rcl}
\mathcal{E} & ::= & [\cdot] \mid \bullet\mathcal{E} \mid \mathcal{E}\ N \mid v\ \mathcal{E} \mid \texttt{fst}\ \mathcal{E} \mid \texttt{snd}\ \mathcal{E} \mid (\mathcal{E}, N) \mid (v, \mathcal{E}) \\
& \mid & (\texttt{case}\ \mathcal{E}\ \texttt{of inl}\ a \mapsto N_1 \ \mid \texttt{inr}\ b \mapsto N_2) \\
& \mid & \texttt{inl}\ \mathcal{E} \mid \texttt{inr}\ \mathcal{E} \mid \texttt{let}\ \bullet x = \mathcal{E}\ \texttt{in}\ N \mid \texttt{inj}\ \mathcal{E} \\
& \mid & \texttt{rec}^T(\vec{u}; rf.M)\ \vec{C}\ \mathcal{E} \mid \texttt{out}\ \mathcal{E} \mid \texttt{corec}^T(\vec{u}; rf.M)\ \vec{C}\ \mathcal{E} \\
& \mid & \mathcal{E}\ \vec{C} \mid \texttt{pack}(C, \mathcal{E}) \mid \texttt{let pack}(u, x) = \mathcal{E}\ \texttt{in}\ N \\
& \mid & \texttt{eq}\ \mathcal{E}_{C_1 = C_2}\ \texttt{with}\ (\Delta; \theta \mapsto N)
\end{array}$$

Figure 3–49: Congruence Rules

rule in Figure 3–48. The congruence rules for Jackrabbit are given by the evaluation contexts in Figure 3–49.

In our setting, we are interested in reduction under $\bullet$. Similarly to $\alpha$-values, we index terms by a time step $k$, written $M^k$ and defined in Figure 3–50. Cave et al. [2014] use evaluation contexts indexed by a temporal depth, $k$. In our presentation, we separate the temporal index from the evaluation context. This separation allows $\bullet$ in the body of functions without also allowing evaluation under binders. We also note that the next-step modality is not allowed in the body of (co)recursive functions, since it is possible to recurse over a temporal object and the body is required at each step. We use the rule in Figure 3–51 to create evaluation stepping from our small-step semantics. At term can step as long as it has permission to move $\alpha$ steps forward and neither term has a depth greater than $\alpha$.

## 3.9 Formal Languages

In addition to the correspondence between types and proofs, we exploit the correspondence between LTL propositions and formal languages. An $\omega$-regular language

$$
\begin{array}{lcl}
M^{k+1} & ::= & \bullet M^k \\
M^{max(j,k)} & ::= & M^j N^k \mid \mathtt{let\ pack}(u,x) = M^j\ \mathtt{in}\ N^k \\
 & \mid & (M^j, N^k) \mid \mathtt{eq}\ {M^j}_{C_1=C_2}\ \mathtt{with}\ (\Delta'; \rho' \mapsto N^k) \\
 & \mid & \mathtt{let}\ \bullet x = M^j\ \mathtt{in}\ N^k \\
M^{max(j,k,l)} & ::= & (\mathtt{case}\ M^j\ \mathtt{of}\ \mathtt{inl}\ a \mapsto N_0^k\ \mid \mathtt{inr}\ b \mapsto N_1^l) \\
M^k & ::= & \lambda x.M^k \mid \mathtt{fst}\ M^k \mid \mathtt{snd}\ M^k \mid \mathtt{inl}\ M^k \mid \mathtt{inr}\ M^k \\
 & \mid & \mathtt{rec}^A(\vec{u}; rf.M^0)\ \vec{C}\ N^k \mid \mathtt{inj}\ M^k \\
 & \mid & \mathtt{corec}^A(\vec{u}; rf.M^0)\ \vec{C}\ N^k \mid \mathtt{out}\ M^k \\
 & \mid & \Lambda u.M^k \mid \mathtt{eqabort}^A_{C_1 \neq C_2}\ M^k \mid M^k\ C \mid \mathtt{pack}(C, M^k) \\
M^0 & ::= & x \mid () \mid \mathtt{refl}
\end{array}
$$

Figure 3–50: $\alpha$-Terms

$$
\frac{M \longrightarrow N}{\mathcal{E}[M]^m \rightsquigarrow_\alpha \mathcal{E}[N]^m}\ \text{where } m \leq \alpha
$$

Figure 3–51: Temporal stepping

| | | | |
|---|---|---|---|
| Regular Expressions | $r_1, r_2$ | $::=$ | $\emptyset \mid \epsilon \mid a \in \Sigma \mid r_1 \cdot r_2 \mid r_1 + r_2 \mid r_1*$ |
| $\omega$-regular Expressions | $s_1, s_2$ | $::=$ | $r^\omega \mid r_1 \cdot s_1 \mid s_1 + s_2$, where $r$ is a regular expression without $\emptyset$ or $\epsilon$ |

Figure 3–52: Regular and $\omega$-regular Expressions

$$\begin{array}{rcl}\Box A & \equiv & \nu Z.A \times \bigcirc Z \\ \Diamond A & \equiv & \mu Z.A + \bigcirc Z \\ A \,\mathcal{U}\, B & \equiv & \mu Z.B + A \times \bigcirc Z \\ A \,\hat{\mathcal{U}}\, B & \equiv & A \times \bigcirc(A \,\hat{\mathcal{U}}\, B)\end{array}$$

Figure 3–53: Defining Temporal Types

is a language for infinite words, characterized by a Büchi automaton. Rather than accepting a string after reaching an accept state as is the case for DFAs and NFAs, an infinite string must hit an accept state infinitely often in a Büchi automaton to be accepted. Cave et al. [2014] point out the correspondence between types in their language and Büchi automata.

As with formal languages, we note that the power of our system comes not from writing programs that could not be written before, but rather by using types that constrain programs in ways that were not possible before. The $\omega$-regular expression $(a + b)^\omega$ allows any possible combination of $a$'s and $b$'s. The power of the system in Cave et al. [2014] is derived from the fact that it's possible to reject every program with a finite number of $a$'s or a finite number of $b$'s.

Recall the temporal types defined in 3–53. The $\omega$-regular expression (defined in Figure 3–52) for fair interleavings is $(a * bb * a)^\omega$, and corresponds to the type $\nu Z.(A \;\mathcal{U}\; (B \;\hat{\mathcal{U}}\; (A \times \bigcirc Z)))$. There are many properties that $\omega$-regular expression cannot encode, however. In particular, it cannot encode those that use past values as a means to restrict current and future values.

**Example 6** (Bounded Streams)**.** Although the previous type is called a fair stream, it allows for one process to be served much more often than the other. The only constraint is that we *eventually* see each of them again. The proportion of $a$'s to $b$'s

$$\text{BoundStream}\ A\ B \equiv \nu Z.\Lambda u.\ (B \times \bigcirc Z(\text{suc}\ u)) + (\Sigma v.u = \text{suc}\ v \times (A \times \bigcirc Z v))$$

Figure 3–54: BoundStream Type

$$\begin{array}{l}
all\_B : \Box A \to \Box B \to BoundStreamA\ B\ \text{zero} \\
all\_B \equiv \lambda as.\lambda bs. \\
\texttt{corec}^{\Box A \times \Box B \to \text{BoundStream}\ A\ B\ \text{zero}}(u; rpf. \\
\quad \texttt{let}\ bs = \texttt{snd}\ r\ \texttt{in} \\
\quad \texttt{let}\ \bullet\, bs' = \texttt{snd}\ (\texttt{out}\ bs)\ \texttt{in} \\
\quad \texttt{inl}\ (\texttt{fst}\ (\texttt{out}\ bs), \bullet f\ (\text{suc}\ u)\ bs') \\
\quad ) \\
\text{zero}\ (as, bs)
\end{array}$$

Figure 3–55: A Stream of ‘b’s

could tend towards infinity since the proportion of one type of object to the other is unbounded. We start by considering the restriction that at every point in time, there are no more $a$s than $b$s. We take as input a stream of values $a : A$ and $b : B$, and produce an interleaving of these values with the invariant that $|a| \leq |b|$. This is a non-regular safety property – non-regular because it cannot be expressed with a Büchi automaton, and a safety property because for every string that should not be accepted, there is some finite prefix that acts as a witness as to why we should reject that string. In particular, that prefix will contain more $a$'s than $b$'s. The type of this interleaver is shown in Figure 3–54.

We treat the indices as a simple kind of memory and use them to count the difference between the number of $b$'s and the number of $a$'s we've seen. In the following term, we use the let construct rather than writing the term inlined to make this example more readable, where `let` $x = N$ `in` $M$ is equivalent to $[N/x]M$. On one end of the spectrum, we can make a stream with only $b$'s in figure 3–55.

```
$alt\_AB : \Box A \rightarrow \Box B \rightarrow BoundStreamA\ B$ zero
$alt\_AB \equiv \lambda as.\lambda bs.$
corec$^{(\Box A \times \Box B)\times \texttt{bool} \rightarrow \text{BoundStream } A\ B \text{ zero}}(u; rpf.$
   let $a$ = fst (out (fst (fst $r$))) in
   let $b$ = fst (out (snd (fst $r$))) in
   let $\bullet as'$ = snd (out (fst (fst $r$))) in
   let $\bullet bs'$ = snd (out (snd (fst $r$))) in
   case (snd $r$) of
     inl $fls \mapsto$ inl $(b, \bullet f$ (suc $u$) $((as', bs'),$ true$))$
   | inr $tru \mapsto$ inr (pack(zero, (refl, $(a, \bullet f$ zero $((as', bs'),$ false$))))))$
zero $((as, bs),$ false$)$
```

Figure 3–56: An Alternating Stream

On the other extreme is the stream with an $a$ whenever possible - the alternating stream “babababab...” in figure 3–56. We provide a boolean as a means of switching between an $a$ and a $b$, where the boolean is either `inl` $fls$ or `inr` $tru$ (where $tru = fls = ()$).

Both of these programs decide beforehand when to switch, but the decision can also be made elsewhere. Instead of taking a single boolean, it’s possible to take a stream of booleans that tells the program when to serve an $a$. An $a$ can then be served when it’s possible, and a $B$ when it’s not. This can get complicated since we cannot pattern match on index objects, although it is possible to propagate and case analyze a vector.

As with regular languages, any program that we write here can also be written without an indexed language. The strength of our type system is that it can draw a line between accepted and unaccepted programs in a way that an indexed language

Number of $a$'s

| | $(m,n)$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | $\cdots$ |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | (1,0) | (0,1) | ∅ | ∅ | ∅ | ∅ | ∅ | $\cdots$ |
| | 2 | ∅ | (2,0) | (1,1) | (0,2) | ∅ | ∅ | ∅ | $\cdots$ |
| Number | 3 | ∅ | ∅ | (3,0) | (2,1) | (1,2) | (0,3) | ∅ | $\cdots$ |
| of $b$'s | 4 | ∅ | ∅ | ∅ | (4,0) | (3,1) | (2,2) | (1,3) | $\cdots$ |
| | 5 | ∅ | ∅ | ∅ | ∅ | (5,0) | (4,1) | (3,2) | $\cdots$ |
| | 6 | ∅ | ∅ | ∅ | ∅ | ∅ | (6,0) | (5,1) | $\cdots$ |
| | 7 | ∅ | ∅ | ∅ | ∅ | ∅ | ∅ | (7,0) | $\cdots$ |
| | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\ddots$ |

Figure 3–57: Index Pairs $(m,n)$ Relative to $a$'s and $b$'s Seen

cannot replicate. Any attempt would either accept programs that it should reject or reject programs that it should accept.

**Example 7** (Bounding at both ends)**.** In the example $|a| \leq |b|$, we know that there will never be fewer $b$'s than $a$'s, but there is no guarantee that $A$ will eventually be served. We can restore fairness by adding a lower bound on the number of $a$'s, for example in a stream where $|b| \leq |a| \leq 2|b|$.

$$(\nu Z.\Lambda m{:}\mathbb{N}, n{:}\mathbb{N}.(\Sigma u.n = \text{suc (suc } u) \times (B \times \bigcirc Z \text{ (suc (suc } m))) \ u)$$
$$+(\Sigma u.m = \text{suc } u \times (A \times \bigcirc Z \ u \text{ (suc (suc } n))))$$
$$\text{(suc zero) (suc zero)}$$

We can think of the first index, $m$, as the number of $a$ permissions, and the second index, $n$, as the number of $b$ permissions. The $b$ permissions work as in the previous example, incrementing every time we see an $a$ and decrementing every time we see a $b$. The $a$ permissions work similarly, but now, every time we see a $b$, we get two $a$ permissions. Since our index language consists of natural numbers, $m$ and $n$ cannot be negative.

In figure 3–57, we write permissions as pairs, where $m$ is the first element and $n$ is the second. For pairs where either index is less than 0, that is, unreachable states, we write ∅. Seeing an $a$ is a step to the right and seeing a $b$ is a step down. We start with one $a$ permission and assuming we've seen one $a$ and one $b$ because the invariant cannot hold otherwise.

**Example 8** (Using BoundStreams)**.** In the previous example, we created an object with a nice type. Creating a stream with an embedded constraint gives an idea of the nature of a BoundStream, but to use it to its full potential, we create a function that takes a BoundStream as a parameter. We can then make greater assumptions about this input. As a simple example, we can create a consumer that acts like a bank account. For every unitary withdrawal ($W$), there must be a corresponding deposit ($D$). We don't need any mechanism to handle more withdrawals than deposits because the type of a BoundStream ensures that this will never happen. As an output, we assume the existence of a function `apply` that takes a withdrawal or deposit and produces a balance of type $B$, which we record as a stream. Although we drop the index $u$, it could be passed forward so as to index the balance stream, or to the `apply` function acting as a certificate. In the case of withdrawal, `fst` $w$ provides the proof that $u$ is not zero. We recall the type of a boundstream here.

$$\text{BoundStream } A\ B \equiv \nu Z.\Lambda u.\ (B \times \bigcirc Z(\text{suc } u)) + (\Sigma v.u = \text{suc } v \times (A \times \bigcirc Z v))$$

**Example 9** (Echo)**.** Recall that with fair interleavings, we know that each process will be served at some point, but we know nothing about when that might be. Indices can place tighter constraints here. Without an indexed type system, we can write a

```
$consumer \equiv BoundStream\ W\ D\ \text{zero} \rightarrow \Box B$
corec$^{\text{BoundStream } W\ D \text{ zero}\rightarrow\Box B}(u; rpf.$
   case (out $r$) of
     inl $d \mapsto$ let $\bullet\, bs' =$ snd $d$ in
                (apply fst $d, \bullet f\ bs'$)
  | inr $w \mapsto$ let $\bullet\, bs' =$ snd (snd $w$) in
                (apply (fst (snd $w$)), $\bullet f\ bs'$))
zero $bs$
```

Figure 3–58: Using a BoundStream

type that ensures that an $A$ is available in $C$ steps, for any constant $C$. With the indexed type system, however, $C$ need not be constant. In the following example, the type ensures that $A$ be served in no longer than the amount of time it took to serve $B$. We therefore wait $t$ steps for $B$ to be served, and then start counting down from $t$ until an $A$ must be served. To make this type more readable, we first define the type "$A$ in at most $t$ steps":

$\Diamond A^{\leq t} \equiv (\mu Z.\Lambda n.\ A + (\Sigma m.n = \text{suc } m \times \bigcirc Z m))\ t$

Now that we have a type corresponding to a countdown indexed by $t$, we can count up $t$ before starting the countdown. This is similar to a vector where the nil case is replaced by a $\Diamond A^{\leq t}$:

$\text{Echo} \equiv (\mu Y.\Lambda u.\ (\bigcirc Y(\text{suc } u)) + (B \times \Diamond A^{\leq u})\ \text{zero})$

# CHAPTER 4
# Metatheory

Strong foundations make it easier to reason about a language. If a language is properly constrained, the same techniques that the language uses to tackle problems can be applied to the language itself. In particular, fixed points allow Jackrabbit to write programs with inductive structures, but since our language is itself defined inductively, we can also use induction to reason about Jackrabbit.

We consider subject reduction and progress. Subject reduction means that a program's type is preserved under evaluation. If the program `fst` $(a, b)$ has type $A$, and `fst` $(a, b) \longrightarrow a$ then $a$ must also have type $A$. The shape of the output is therefore known at compile time. Progress ensures that a program won't get stuck. Every valid program must either be fully evaluated, or there must be a stepping rule that takes it to another valid program. To prove subject reduction of Jackrabbit, we require additional lemmas.

## 4.1 Substitution Lemmas

Substitution lemmas state that a program's type is preserved under a well-typed substitution. That is, if $N$ and $M$ are well-typed, then so is $[N/x]M$, and it will have the same type as $M$. We use substitution lemmas for each of the term substitutions: regular variable substitutions, future substitutions, index substitutions, and omega substitutions. We present the more interesting cases here.

**Lemma 1** (Index substitution lemma)**.** *If* $\Delta; \Theta; \Gamma \vdash M : A$ *and* $\Delta'' \vdash \sigma : \Delta$ *then* $\Delta''; \{\sigma\}\Theta; \{\sigma\}\Gamma \vdash \{\sigma\}M : \{\sigma\}A$

As a corollary, we have that if $\Delta, u{:}U; \Theta; \Gamma \vdash M : A$ and $\Delta \vdash C : U$ then $\Delta; \{C/u\}\Theta; \{C/u\}\Gamma \vdash \{C/u\}M : \{C/u\}A$

The most interesting case is that of equality elimination. This rule makes it possible to substitute index objects in types, yet substitution lemmas are about proving the type is invariant. We can do this because index objects in types are only allowed to "change" when they are equal. We know that $C_1$ and $C_2$ unify, but we don't have this guarantee for $\{\sigma\}C_1$ and $\{\sigma\}C_2$. We therefore consider both cases. If they do unify, then we can use their most general unifier and the inductive hypothesis to construct the required type. If they do not unify, we get $\mathtt{eqabort}^{\{\sigma\}A}_{C_1 \neq C_2}\ M$ which provides the necessary type. This happens in uninhabited branches, such as in the nil case of the tail of a non-empty vector.

*Proof.* By induction on the typing derivation $\mathcal{D} = \Delta; \Theta; \Gamma \vdash M : T$.

Case: $\mathcal{D} =$

$$\frac{\Delta; \Theta; \Gamma \vdash M : C_1 = C_2 \qquad \begin{array}{c}\Delta \vdash \mathtt{unify}(C_1, C_2) \Rightarrow (\theta; \Delta') \\ \Delta'; \{\theta\}\Theta; \{\theta\}\Gamma \vdash N : \{\theta\}A\end{array}}{\Delta; \Theta; \Gamma \vdash \mathtt{eq}\ M_{C_1 = C_2}\ \mathtt{with}\ (\Delta'; \theta \mapsto N) : A} = E_1$$

$\Delta; \Theta; \Gamma \vdash M : C_1 = C_2$ By inversion

$\Delta''; \{\sigma\}\Theta; \{\sigma\}\Gamma \vdash \{\sigma\}M : \{\sigma\}C_1 = \{\sigma\}C_2$ By I.H.

$\Delta \vdash \mathtt{unify}(C_1, C_2) \Rightarrow (\theta; \Delta')$ By inversion

$\Delta'' \vdash \sigma : \Delta$ By assumption

$\Delta'' \vdash \texttt{unify}(\{\sigma\}C_1, \{\sigma\}C_2) \Rightarrow (\theta_1; \Delta_1)$ Subcase 1

$\exists\ \theta_2\ s.t.\ \Delta_1 \vdash \{\theta_2\}\theta : \Delta\ \&\ \Delta_1 \vdash \theta_2 : \Delta'$ Since $\texttt{mgu}\ \theta$

$\Delta'; \{\theta\}\Theta; \{\theta\}\Gamma \vdash N : \{\theta\}A$ By inversion

$\Delta_1; \{\theta_2\}\{\theta\}\Theta; \{\theta_2\}\{\theta\}\Gamma \vdash \{\theta_2\}N : \{\theta_2\}\{\theta\}A$ By I.H. with $\theta_2$

$\Delta_1; \{\theta_1\}\{\sigma\}\Theta; \{\theta_1\}\{\sigma\}\Gamma \vdash \{\theta_2\}N : \{\theta_1\}\{\sigma\}A$ Since $\{\theta_2\}\theta = \{\theta_1\}\sigma$

$\Delta''; \{\sigma\}\Theta; \{\sigma\}\Gamma \vdash \texttt{eq}\ \{\sigma\}M_{C_1=C_2}\ \texttt{with}\ (\Delta_1; \theta_1 \mapsto \{\theta_2\}N) : \{\sigma\}A$ By $=E_1$

$\Delta''; \{\sigma\}\Theta; \{\sigma\}\Gamma \vdash \{\sigma\}\texttt{eq}\ M_{C_1=C_2}\ \texttt{with}\ (\Delta'; \theta \mapsto N) : \{\sigma\}A$ By definition of substitution

$\Delta'' \vdash \texttt{unify}(\{\sigma\}C_1, \{\sigma\}C_2) \Rightarrow \bot$ Subcase 2

$\Delta''; \{\sigma\}\Theta; \{\sigma\}\Gamma \vdash \{\sigma\}\texttt{eqabort}^{\{\sigma\}A}_{C_1 \neq C_2}\ M : \{\sigma\}A$ By definition of substitution

□

**Lemma 2** (Regular Substitution Lemma)**.** *If* $\Delta; \Theta; \Gamma, x{:}S \vdash M : T$ *and* $\Delta; \Theta; \Gamma \vdash M' : S$ *then* $\Delta; \Theta; \Gamma \vdash [M'/x]M : T$*.*

Due to the inductive hypothesis, we can assume that the substitution lemma holds for the premises to show that it holds for the conclusion. For equality elimination, we also use the index substitution lemma. When substituting under a bullet, $\Gamma$ is dropped. We therefore don't use the inductive hypothesis in the $\bigcirc$ introduction case.

*Proof.* We prove this by induction on the typing derivation $\mathcal{D} = \Delta; \Theta; \Gamma, x{:}S \vdash M : T$.

Case: $\mathcal{D} = \dfrac{\Delta;\Theta;\Gamma,x{:}S \vdash M : A \qquad \Delta;\Theta;\Gamma,x{:}S \vdash N : B}{\Delta;\Theta;\Gamma,x{:}S \vdash (M,N) : A \times B}\times I$

$\Delta;\Theta;\Gamma \vdash M' : S$ By assumption

$\Delta;\Theta;\Gamma \vdash [M'/x]M : A$ By I.H.

$\Delta;\Theta;\Gamma \vdash [M'/x]N : B$ By I.H.

$[M'/x](M,N) = ([M'/x]M,[M'/x]N)$ By definition of substitution

$\Delta;\Theta;\Gamma \vdash [M'/x](M,N) : A \times B$ By definition of ×

Case: $\mathcal{D} = \dfrac{\Delta;\Theta;\Gamma,x{:}S,y{:}A \vdash M : B}{\Delta;\Theta;\Gamma,x{:}S \vdash \lambda y.M : A \to B}\to I$

$\Delta;\Theta;\Gamma \vdash M' : S$ By assumption

$\Delta;\Theta;\Gamma,y{:}A \vdash [M'/x]M : B$ By I.H. (exchange and assuming $y \neq x$ & $y \notin \mathtt{FV}(M')$)

$\Delta;\Theta;\Gamma \vdash \lambda y.[M'/x]M : A \to B$ By $\to I$

$\Delta;\Theta;\Gamma \vdash [M'/x]\lambda y.M : A \to B$ By definition of substitution

Case: $\mathcal{D} =$

$$\dfrac{\Delta;\Theta;\Gamma,x{:}S \vdash M : C_1 = C_2 \qquad \begin{array}{c}\Delta \vdash \mathtt{unify}(C_1,C_2) \Rightarrow (\theta;\Delta') \\ \Delta';\{\theta\}\Theta;\{\theta\}\Gamma,x{:}S \vdash N : \{\theta\}A\end{array}}{\Delta;\Theta;\Gamma,x{:}S \vdash \mathtt{eq}\ M_{C_1=C_2}\ \mathtt{with}\ (\Delta';\theta \mapsto N) : A} {=}E_1$$

$\Delta;\Theta;\Gamma \vdash M' : S$ By assumption

$\Delta';\{\theta\}\Theta;\{\theta\}\Gamma \vdash \{\theta\}M' : \{\theta\}S$ By index substitution lemma

$\Delta;\Theta;\Gamma \vdash [M'/x]M : C_1 = C_2$ By I.H.

$\Delta';\{\theta\}\Theta;\{\theta\}\Gamma \vdash [\{\theta\}M'/x]N : \{\theta\}A$ By I.H.

$\Delta;\Theta;\Gamma \vdash \mathtt{eq}\ [M'/x]M_{C_1=C_2}\ \mathtt{with}\ (\Delta';\theta \mapsto [\{\theta\}M'/x]N) : A$ By $= E_1$

$\Delta;\Theta;\Gamma \vdash [M'/x]\ \mathtt{eq}\ M_{C_1=C_2}\ \mathtt{with}\ (\Delta';\theta \mapsto N) : A$ By definition of substitution

Case: $\mathcal{D} = \dfrac{\Delta; \cdot; \Theta \vdash M : A}{\Delta; \Theta; \Gamma, x{:}S \vdash \bullet M : \bigcirc A} \bigcirc I$

$\Delta; \Theta; \Gamma \vdash M' : S$ By assumption

$\Delta; \Theta; \Gamma \vdash [M'/x] \bullet M : \bigcirc A$ By inversion, $\bigcirc I$, definition of substitution

Case: $\mathcal{D} = \dfrac{\Delta; \Theta; \Gamma, x{:}S \vdash M : \bigcirc A \qquad \Delta; \Theta, y{:}A; \Gamma, x{:}S \vdash N : B}{\Delta; \Theta; \Gamma, x{:}S \vdash \texttt{let } \bullet y = M \texttt{ in } N : B} \bigcirc E$

$\Delta; \Theta; \Gamma \vdash M' : S$ By assumption

$\Delta; \Theta; \Gamma \vdash [M'/x]M : \bigcirc A$ By I.H.

$\Delta; \Theta, y{:}A; \Gamma \vdash [M'/x]N : B$ By I.H.

$\Delta; \Theta; \Gamma \vdash \texttt{let } \bullet y = [M'/x]M \texttt{ in } [M'/x]N : B$ By $\bigcirc I$

$\Delta; \Theta; \Gamma \vdash [M'/x] \texttt{ let } \bullet y = M \texttt{ in } N : B$ By definition of substitution

□

**Lemma 3** (Future Substitution Lemma)**.** *If* $\Delta; \Theta, x{:}S; \Gamma \vdash M : T$ *and* $\Delta; \Theta; \Gamma \vdash \bullet N : \bigcirc S$ *then* $\Delta; \Theta; \Gamma \vdash [N/x]^{\bullet} M : T$*.*

This is similar to the substitution lemma. In the $\bigcirc$ introduction case, we appeal to the substitution lemma rather than the inductive hypothesis.

*Proof.* By induction on the typing derivation $\mathcal{D} = \Delta; \Theta, x{:}S; \Gamma \vdash M : T$.

Case: $\mathcal{D} = \dfrac{\Delta; \Theta, x{:}S; \Gamma \vdash M : A \to B \qquad \Delta; \Theta, x{:}S; \Gamma \vdash N : A}{\Delta; \Theta, x{:}S; \Gamma \vdash MN : B} \to E$

$\Delta; \Theta; \Gamma \vdash \bullet M' : \bigcirc S$ By assumption

$\Delta; \Theta; \Gamma \vdash [M'/x]^{\bullet} M : A \to B$ By I.H.

$\Delta; \Theta; \Gamma \vdash [M'/x]^{\bullet} N : A$ By I.H.

$\Delta; \Theta; \Gamma \vdash ([M'/x]^{\bullet} M)([M'/x]^{\bullet} N) : B$ By $\to E$

$\Delta; \Theta; \Gamma \vdash [M'/x]^{\bullet}\ MN : B$ By definition of future substitution

Case: $\mathcal{D} = \dfrac{\Delta; \cdot; \Theta, x{:}S \vdash M : A}{\Delta; \Theta, x{:}S; \Gamma \vdash \bullet M : \bigcirc A}\,\bigcirc I$

$\Delta; \Theta; \Gamma \vdash \bullet M' : \bigcirc S$ By assumption

$\Delta; \cdot; \Theta \vdash [M'/x]M : A$ By inversion, substitution lemma

$\Delta; \Theta; \Gamma \vdash \bullet [M'/x]M : \bigcirc A$ By $\bigcirc I$

$\Delta; \Theta; \Gamma \vdash [M'/x]^{\bullet} \bullet M : \bigcirc A$ By definition of future substitution

Case: $\mathcal{D} = \dfrac{\Delta; \Theta, x{:}S; \Gamma \vdash M : \bigcirc A \qquad \Delta; \Theta, x{:}S, y{:}A; \Gamma \vdash N : B}{\Delta; \Theta, x{:}S; \Gamma \vdash \texttt{let}\ \bullet y = M\ \texttt{in}\ N : B}\,\bigcirc E$

$\Delta; \Theta; \Gamma \vdash \bullet M' : \bigcirc S$ By assumption

$\Delta; \Theta; \Gamma \vdash [M'/x]^{\bullet} M : \bigcirc A$ By I.H.

$\Delta; \Theta, y{:}A; \Gamma \vdash \bullet M' : \bigcirc S$ By weakening

$\Delta; \Theta, y{:}A; \Gamma \vdash [M'/x]^{\bullet} N : B$ By I.H. (assuming $y \neq x$ & $y \notin \texttt{FV}(M')$)

$\Delta; \Theta; \Gamma \vdash \texttt{let}\ \bullet y = [M'/x]^{\bullet} M\ \texttt{in}\ [M'/x]^{\bullet} N : B$ By $\bigcirc I$

$\Delta; \Theta; \Gamma \vdash [M'/x]^{\bullet}\ \texttt{let}\ \bullet y = M\ \texttt{in}\ N : B$ By definition of future substitution

□

**Lemma 4** (Omega Substitution Lemma)**.** *If* $\Delta, x{:}S; \Theta; \Gamma \vdash M : T$ *and* $\Delta; \Theta; \Gamma \vdash M' : S$ *then* $\Delta; \Theta; \Gamma \vdash [M'/x]^{\omega} M : T$.

*Proof.* For every case except for the $\bigcirc$ modality, this is similar to the substitution lemma, with omega substitution rather than regular substitution.

Case: $\mathcal{D} = \dfrac{(\Phi; \Psi; \Omega, x{:}S); \cdot; \Theta \vdash M : A}{(\Phi; \Psi; \Omega, x{:}S); \Theta; \Gamma \vdash \bullet M : \bigcirc A}\,\bigcirc I$

$(\Phi; \Psi; \Omega); \Theta; \Gamma \vdash M' : S$ By assumption

$(\Phi;\Psi;\Omega);\cdot;\Theta \vdash [M'/x]^{\omega} M : A$ By inversion, I.H.

$(\Phi;\Psi;\Omega);\Theta;\Gamma \vdash \bullet[M'/x]^{\omega} M : \bigcirc A$ By $\bigcirc I$

$(\Phi;\Psi;\Omega);\Theta;\Gamma \vdash [M'/x]^{\omega} \bullet M : \bigcirc A$ By definition of omega substitution

□

## 4.2 Subject Reduction

We prove that the stepping relation $\rightsquigarrow_{\alpha}$ preserves type by induction on $M \rightsquigarrow_{\alpha} M'$. For example, $(\lambda x.M)N$ steps to $[N/x]M$. We need to show that the term $(\lambda x.M)N$ has the same type as the term it steps to $[N/x]M$. To do this, we use small-step type preservation as a lemma, which is in turn shown by induction on the stepping derivation $\mathcal{D} = M \longrightarrow M'$. Recall that $\rightsquigarrow_{\alpha}$ is a stepping relation that takes into account timesteps, and is defined using the small-step relation $\longrightarrow$.

**Lemma 5** (Small-step subject reduction)**.** *If* $\Delta;\Theta;\Gamma \vdash M : A$ *and* $M \longrightarrow M'$ *then* $\Delta;\Theta;\Gamma \vdash M' : A$.

*Proof.* Case: $\mathcal{D} = (\lambda x.M)N \longrightarrow [N/x]M$

$\Delta;\Theta;\Gamma \vdash (\lambda x.M)N : B$ Assumption

$\Delta;\Theta;\Gamma \vdash N : A$ By inversion $\rightarrow E$

$\Delta;\Theta;\Gamma \vdash \lambda x.M : A \rightarrow B$ By inversion $\rightarrow E$

$\Delta;\Theta;\Gamma, x{:}A \vdash M : B$ By inversion $\rightarrow I$

$\Delta;\Theta;\Gamma \vdash [N/x]M : B$ By regular substitution lemma

Case: $\mathcal{D} = \texttt{fst}\ (M,N) \longrightarrow M$

$\Delta;\Theta;\Gamma \vdash \texttt{fst}\ (M,N) : A$ Assumption

$\Delta; \Theta; \Gamma \vdash (M, N) : A \times B$ By inversion $\times E$ (arbitrary $B$)

$\Delta; \Theta; \Gamma \vdash M : A$ By inversion $\times I$

Case: $\mathcal{D} = \texttt{snd}\ (M, N) \longrightarrow N$

Similar to previous case.

Case: $\mathcal{D} = \texttt{case}\ (\texttt{inl}\ M)\ \texttt{of}\ \texttt{inl}\ x \mapsto N_1\ \mid \texttt{inr}\ y \mapsto N_2 \longrightarrow [M/x]N_1$

$\Delta; \Theta; \Gamma \vdash \texttt{case}\ (\texttt{inl}\ M)\ \texttt{of}\ \texttt{inl}\ x \mapsto N_1\ \mid \texttt{inr}\ y \mapsto N_2 : T$ Assumption

$\Delta; \Theta; \Gamma \vdash \texttt{inl}\ M : A + B$ By inversion $+E$

$\Delta; \Theta; \Gamma \vdash M : A$ By inversion $+I$

$\Delta; \Theta; \Gamma, x : A \vdash N_1 : T$ By inversion

$\Delta; \Theta; \Gamma \vdash [M/x]N_1 : T$ By regular substitution lemma

Case: $\mathcal{D} = \texttt{case}\ (\texttt{inr}\ M)\ \texttt{of}\ \texttt{inl}\ x \mapsto N_1\ \mid \texttt{inr}\ y \mapsto N_2 \longrightarrow [M/y]N_2$

Similar to previous case.

Case: $\mathcal{D} = \texttt{let}\ \bullet x = \bullet M\ \texttt{in}\ N \longrightarrow [M/x]^{\bullet}N$

$\Delta; \Theta; \Gamma \vdash \texttt{let}\ \bullet x = \bullet M\ \texttt{in}\ N : T$ Assumption

$\Delta; \Theta, x : A; \Gamma \vdash N : T$ By inversion $\bullet E$

$\Delta; \Theta; \Gamma \vdash \bullet M : \bigcirc A$ By inversion $\bullet E$

$\Delta; \Theta; \Gamma \vdash [M/x]^{\bullet}N : T$ By future substitution lemma

Case: $\mathcal{D} = (\Lambda u.M)C \longrightarrow \{C/u\}M$

| | |
|---|---|
| $\Delta; \{C/u\}\Theta; \{C/u\}\Gamma \vdash (\Lambda u.M)C : \{C/u\}A$ | By assumption, inversion $\Pi E$ |
| $\Delta; \Theta; \Gamma \vdash C : U$ | By inversion $\Pi E$ |
| $\Delta; \Theta; \Gamma \vdash \Lambda u.M : \Pi u{:}U.A$ | By inversion $\Pi E$ |
| $\Delta, u{:}U; \Theta; \Gamma \vdash M : A$ | By inversion $\Pi I$ |
| $\Delta \vdash C/u : \Delta, u{:}U$ | By inspection |
| $\Delta; \{C/u\}\Theta; \{C/u\}\Gamma \vdash \{C/u\}M : \{C/u\}A$ | By index substitution lemma |

Case: $\texttt{let}\ \texttt{pack}(u, x) = \texttt{pack}(C, M)\ \texttt{in}\ N \longrightarrow [M/x]\{C/u\}N$

| | |
|---|---|
| $\Delta; \Theta; \Gamma \vdash \texttt{let}\ \texttt{pack}(u, x) = \texttt{pack}(C, M)\ \texttt{in}\ N : B$ | Assumption |
| $\Delta, u{:}U; \Theta; \Gamma, x : A \vdash N : B$ | By inversion $\Sigma E$ |
| $\Delta; \Theta; \Gamma \vdash \texttt{pack}(C, M) : \Sigma u{:}U.A$ | By inversion $\Sigma E$ |
| $\Delta \vdash C : U$ | By inversion $\Sigma I$ |
| $\Delta; \Theta; \Gamma \vdash M : \{C/u\}A$ | By $\Sigma I$ |
| $\Delta; \{C/u\}\Theta; \{C/u\}(\Gamma, x{:}A) \vdash \{C/u\}N : \{C/u\}B$ | By index substitution lemma |
| $\Delta; \Theta; \Gamma, x{:}\{C/u\}A \vdash \{C/u\}N : B$ | Since $u$ does not appear in $\Theta$, $\Gamma$, or $B$ |
| $\Delta; \Theta; \Gamma \vdash [M/x](\{C/u\}N) : B$ | By regular substitution lemma |
| $\Delta; \Theta; \Gamma \vdash [M/x]\{C/u\}N : B$ | By definition of substitution |

Case: $\mathcal{D} =$

$\texttt{rec}^B(\vec{u}; rf.M)\ \vec{C}\ (\texttt{inj}\ N) \longrightarrow [N/r][\Lambda\vec{u}.\lambda y.(\texttt{rec}^B(\vec{u}; rf.M)\ \vec{u}\ y)/f]^{\omega}\{\vec{C}/\vec{u}\}M^{[\![R/Z]\!]}$,

where $B = \Pi\overrightarrow{u{:}U}.(\mu Z{:}K.\Lambda\vec{u}.S)\vec{u} \to T$ and $R = \mu Z.\Lambda\overrightarrow{u{:}U}.S$

| | |
|---|---|
| $(\Phi; \Psi); \Theta; \Gamma \vdash \texttt{rec}^B(\vec{u}; rf.M)\ \vec{C}\ (\texttt{inj}\ N) : A$ | Assumption |

$A = \{\vec{C}/\vec{u}\}T$ By inversion $\mu E$

$(\Phi, Z{:}K; \overrightarrow{u{:}U}; f{:}\Pi\vec{v}{:}U.Z\vec{v} \to \{\vec{v}/\vec{u}\}T); \cdot; r{:}S \vdash M : T$ By inversion $\mu E$

$(\Phi; \Psi); \Theta; \Gamma \vdash \texttt{inj}\ N : (\mu Z.\Lambda\overrightarrow{u{:}U}.S)\vec{C}$ By inversion $\mu E$

$(\Phi; \Psi); \Theta; \Gamma \vdash N : [\![R/Z]\!]\{\vec{C}/\vec{u}\}S$ By inversion $\mu I$

$\Psi \vdash \overrightarrow{C{:}U}$ Since $\texttt{inj}$ $N$ is well-typed

$(\Phi, Z{:}K; \cdot; f{:}\Pi\vec{v}{:}U.Z\vec{v} \to \{\vec{v}/\vec{u}\}T); \cdot; \{\vec{C}/\vec{u}\}(r{:}S) \vdash \{\vec{C}/\vec{u}\}M : \{\vec{C}/\vec{u}\}T$ By index substitution lemma

$(\Phi; \cdot; f{:}\Pi\vec{v}{:}U.(\mu Z.\Lambda\overrightarrow{u{:}U}.S)\vec{v} \to \{\vec{v}/\vec{u}\}T); \cdot; r{:}\{\vec{C}/\vec{u}\}S \vdash \{\vec{C}/\vec{u}\}M^{[\![R/Z]\!]} : \{\vec{C}/\vec{u}\}T$ By type substitution

Substituting $f$:

$(\Phi, Z{:}K; \overrightarrow{u{:}U}; f{:}\Pi\vec{v}{:}U.Z\vec{v} \to \{\vec{v}/\vec{u}\}T); \cdot; r{:}S \vdash M : T$ From previous inversion $\mu E$

$(\Phi; \Psi, \overrightarrow{v{:}U}); \Theta; \Gamma, y{:}(\mu Z.\Lambda\vec{u}.S)\vec{v} \vdash y : (\mu Z.\Lambda\vec{u}.S)\vec{v}$ By $xI$, weakening

$(\Phi; \Psi, \overrightarrow{v{:}U}); \Theta; \Gamma, y{:}(\mu Z.\Lambda\vec{u}.S)\vec{v} \vdash \texttt{rec}^B(\vec{v}; rf.M)\ \vec{v}\ y : \{\vec{v}/\vec{u}\}T$ By $\mu E$

$(\Phi; \Psi, \overrightarrow{v{:}U}); \Theta; \Gamma \vdash \lambda y.\texttt{rec}^B(\vec{v}; rf.M)\ \vec{v}\ y : (\mu Z.\Lambda\vec{u}.S)\vec{v} \to \{\vec{v}/\vec{u}\}T$ By $\to I$

$(\Phi; \Psi); \Theta; \Gamma \vdash \Lambda\vec{v}.\lambda y.\texttt{rec}^B(\vec{v}; rf.M)\ \vec{v}\ y : \Pi\overrightarrow{v{:}U}.(\mu Z.\Lambda\vec{u}.S)\vec{v} \to \{\vec{v}/\vec{u}\}T$ By $\Pi I$

$(\Phi; \Psi); \Theta; \Gamma, r{:}\{\vec{C}/\vec{u}\}S \vdash \Lambda\vec{v}.\lambda y.\texttt{rec}^B(\vec{v}; rf.M)\ \vec{v}\ y : \Pi\overrightarrow{v{:}U}.(\mu Z.\Lambda\vec{u}.S)\vec{v} \to \{\vec{v}/\vec{u}\}T$ Weakening

$(\Phi; \cdot; f{:}\Pi\vec{v}{:}U.(\mu Z.\Lambda\overrightarrow{u{:}U}.S)\vec{v} \to \{\vec{v}/\vec{u}\}T); \cdot; r{:}\{\vec{C}/\vec{u}\}S \vdash \{\vec{C}/\vec{u}\}M^{[\![R/Z]\!]} : \{\vec{C}/\vec{u}\}T$ Recall from before substituting $f$

$(\Phi; \Psi; f{:}\Pi\vec{v}{:}U.(\mu Z.\Lambda\overrightarrow{u{:}U}.S)\vec{v} \to \{\vec{v}/\vec{u}\}T); \Theta; r{:}\{\vec{C}/\vec{u}\}S \vdash \{\vec{C}/\vec{u}\}M^{[\![R/Z]\!]} : \{\vec{C}/\vec{u}\}T$ Weakening

let $fsbst = \Lambda\vec{v}.\lambda y.(\texttt{rec}^B(\vec{v}; rf.M)\ \vec{v}\ y)$

$(\Phi;\Psi;\cdot);\Theta;\Gamma, r{:}\{\vec{C}/\vec{u}\}S \vdash [fsbst/f]^{\omega}\{\vec{C}/\vec{u}\}M^{[\![R/Z]\!]} : \{\vec{C}/\vec{u}\}T$ By omega substitution lemma

$(\Phi;\Psi);\Theta;\Gamma \vdash N : [\![R/Z]\!]\{\vec{C}/\vec{u}\}S$ Recall from previous inversion $\mu I$

$(\Phi;\Psi);\Theta;\Gamma \vdash [N/r][fsbst/f]^{\omega}\{\vec{C}/\vec{u}\}M^{[\![R/Z]\!]} : \{\vec{C}/\vec{u}\}T$ By regular substitution lemma

Case: $\mathcal{D} = \texttt{out}\ (\texttt{corec}^{A}(\vec{u}; rf.M)\ \vec{C}\ N) \longrightarrow$
$\{\vec{C}/\vec{u}\}[\Lambda\vec{w}.\lambda y.(\texttt{corec}^{A}(\vec{u}; rf.M)\ \vec{w}\ y)/f]^{\omega}[N/r]M^{[\![R/Z]\!]}$
Similar to previous case.

Case: $\mathcal{D} = \texttt{eq}\,\texttt{refl}_{C_1=C_2}\ \texttt{with}\ (\Delta';\theta \mapsto N) \longrightarrow N$

$\Delta;\Theta;\Gamma \vdash \texttt{eq}\,\texttt{refl}_{C_1=C_2}\ \texttt{with}\ (\Delta';\theta \mapsto N) : A$ By assumption

$\Delta';\{\theta\}\Theta;\{\theta\}\Gamma \vdash N : \{\theta\}A$ By inversion $=E_1$

$\Delta \vdash \texttt{unify}(C_1, C_2) \Rightarrow (\theta;\Delta')$ By inversion $= E_1$

$\Delta;\Theta;\Gamma \vdash \texttt{refl} : C_1 = C_2$ By inversion $= E_1$

$\Delta;\Theta;\Gamma \vdash \texttt{refl} : C_1 = C_1$ By inversion $= I$

$\theta = \texttt{id}$ By definition of $\texttt{unify}(C_1, C_1) \Rightarrow (\theta;\Delta')$

$\Delta;\Theta;\Gamma \vdash N : A$ By definition of identity substitution

□

**Theorem 2** (Subject reduction for $\rightsquigarrow_\alpha$)**.** *For all* $\alpha$*, if* $\Delta; \Theta; \Gamma \vdash M : A$ *and* $M \rightsquigarrow_\alpha M'$ *then* $\Delta; \Theta; \Gamma \vdash M' : A$.

*Proof.* This is proven by induction on $\mathcal{E}[M]^m$, using subject reduction for $\longrightarrow$ as a lemma.

Recall the stepping rule for $\rightsquigarrow_\alpha$.

$$\frac{M \longrightarrow N}{\mathcal{E}[M]^m \rightsquigarrow_\alpha \mathcal{E}[N]^m} \text{ where } m \leq \alpha$$

Case: $m = 0$

Subcase: $\mathcal{E}[M]^m = [\cdot]$

$\Delta; \Theta; \Gamma \vdash \mathcal{E}[M]^m : A$ Assumption

$\Delta; \Theta; \Gamma \vdash \mathcal{E}[M]^m \rightsquigarrow_\alpha \mathcal{E}[N]^m$ Assumption

$\Delta; \Theta; \Gamma \vdash \mathcal{E}[N]^m : A$ By subject reduction for $\longrightarrow$

Subcase: $(\mathcal{E}[M]^m)\ N$

$\Delta; \Theta; \Gamma \vdash (\mathcal{E}[M]^m)\ N : B$ Assumption

$\Delta; \Theta; \Gamma \vdash \mathcal{E}[M]^m \rightsquigarrow_\alpha \mathcal{E}[M']^m$ Assumption

$\Delta; \Theta; \Gamma \vdash N : A$ By inversion $\rightarrow E$

$\Delta; \Theta; \Gamma \vdash \mathcal{E}[M]^m : A \rightarrow B$ By inversion $\rightarrow E$

$\Delta; \Theta; \Gamma \vdash \mathcal{E}[M']^m : A \rightarrow B$ By I.H.

$\Delta; \Theta; \Gamma \vdash (\mathcal{E}[M']^m) N : B$ By $\rightarrow I$

□

## 4.3 Progress

We state progress and provide examples for a few simple cases, although we do not prove progress in general. For some terms to step, the structure of the term is important, rather than simply the type. For example, equality elimination only steps when $M = \texttt{refl}$. A more exhaustive analysis is required.

**Theorem 3** (Progress for $\rightsquigarrow_\alpha$)**.** *If $\Delta; \Theta; \Gamma \vdash M : A$ then either $M$ is a value or there exists $\alpha \in \mathbb{N}$ and a term $M'$ such that $M \rightsquigarrow_\alpha M'$.*

*Proof.* Case: $\mathcal{D} = \dfrac{\Delta; \Theta; \Gamma \vdash M : A \qquad \Delta; \Theta; \Gamma \vdash N : B}{\Delta; \Theta; \Gamma \vdash (M, N) : A \times B} \times I$

Subcase 1: $M$ and $N$ are both values

$\texttt{val}\ (M, N)$ By definition of value

Subcase 2: $\neg\texttt{val}\ M$ but $\texttt{val}\ N$

$M \rightsquigarrow_\alpha M'$ for some $\alpha$ By I.H.

$(M, N) \rightsquigarrow_\alpha (M', N)$ By definition of $\rightsquigarrow_\alpha$

Subcases 3 and 4 are similar to subcase 2

Case: $\mathcal{D} = \dfrac{\Delta; \cdot; \Theta \vdash M : A}{\Delta; \Theta; \Gamma \vdash \bullet M : \bigcirc A} \bigcirc I$

Subcase 1: $\texttt{val}\ M$

$\texttt{val}\ \bullet M$ By definition of value

Subcase 2: $M$ is not a value

$M \rightsquigarrow_{\alpha'} M'$ for some $\alpha'$ By I.H.

Let $\alpha = \alpha' + 1$

$\bullet M \rightsquigarrow_\alpha \bullet M'$ By definition of $\rightsquigarrow_\alpha$

Case: $\mathcal{D} = \dfrac{\Delta;\Theta;\Gamma \vdash M : (\nu Z.\Lambda\overrightarrow{u{:}U}.S)\vec{C}}{\Delta;\Theta;\Gamma \vdash \texttt{out}\ M : [\![\nu Z.\Lambda\overrightarrow{u{:}U}.S/Z]\!]\{\vec{C}/\vec{u}\}S}\,\nu E$

Subcase 1: $\texttt{val}\ M(M = \texttt{corec}^A(\vec{u}; rf.M')\ \vec{C}\ N)$

$\texttt{out}\ (\texttt{corec}^A(\vec{u}; rf.M')\ \vec{C}\ N) \longrightarrow$

$\{\vec{C}/\vec{u}\}[\Lambda\vec{w}.\lambda y.(\texttt{corec}^A(\vec{u}; rf.M')\ \vec{w}\ y)/f]^{\omega}[N/r]M'$

Subcase 2: $M$ is not a value

$M \rightsquigarrow_\alpha M'$ By I.H.

$\texttt{out}\ M \rightsquigarrow_\alpha \texttt{out}\ M'$ By definition of $\rightsquigarrow_\alpha$

□

# CHAPTER 5
# Conclusion

## 5.1 Summary

We presented Jackrabbit, a reactive programming language with indexed types, which corresponds to first-order linear temporal logic under the Curry-Howard correspondence. We provide the syntax and semantics of Jackrabbit along with rules for kinding, typing, and substitution.

Jackrabbit uses Mendler-style recursion. We give an explanation for this, as well as an overview of general recursion, iterators, and primitive recursion, and their relation to one another. Since Mendler-style recursion lacks some of the power of general recursion, Jackrabbit also uses Mendler-style corecursion, allowing it to handle infinite structures. Separating recursion from corecursion also makes is possible to differentiate between least and greatest fix-points, which we use to define the eventually and always modalities and to specify fairness properties. Indexed types, which correspond to logical predicates, are also supported in Jackrabbit. These allow us to specify safety properties that require a simple kind of state.

We explained safety and fairness, but only briefly mentioned liveness. Recall that given an infinite trace of a program, we can tell that a safety property has been violated given some finite prefix. Common safety properties include mutual exclusion and deadlock freedom. With a liveness property, such as such as starvation freedom, any prefix can be extended to an accepted infinite trace. It's never too late to

$$\nu Z.\Lambda m.\Lambda n.(B \times \bigcirc Z(\text{suc } m)(n + m + \text{suc } m)) \;+\; (\Sigma y.n = \text{suc } y \times (A \times \bigcirc Z\ m\ y))$$
$$(\text{suc zero})(\text{suc zero})$$

Figure 5–1: $|a| \leq |b|^2$

satisfy the condition. Consider the characters '(' and ')'. In a valid sequence of parentheses, every closing parenthesis must match an opening parenthesis. When a safety property is violated, there is a certificate that indicates that the property does not hold. Given the trace "()())((...", the prefix "()())" violates the property, because at no point can we have more closing parentheses than opening parentheses. To ensure a valid string, however, we also need to guarantee the liveness property that the sequence will eventually terminate with as many closing parentheses as opening parentheses. Every property, including fairness properties, can be written as the intersection of safety and liveness properties [Alpern and Schneider, 1985]. For example, to prove total correctness, a program must be partially correct and it must terminate. Partial correctness is a safety property and termination is a liveness property. We also note the relationship between least fixed points and liveness, and the relationship between greatest fixed points and safety properties. A liveness property says that eventually something will happen, and a safety property says that something bad will not happen, that is, its negation will always be true.

On top of the properties that can be expressed in types, we considered properties about Jackrabbit, including progress. Finally, with the help of various substitution lemmas, we showed type preservation.

## 5.2 Future Work

The property of normalization, always reducing to a canonical form, is especially important in a language whose type system is meant to be well-behaved with respect to time and space. We do not show that Jackrabbit is strongly normalizing. Addressing and proving normalization in Jackrabbit would be of interest.

The alternating bit protocol and Lamport's bakery protocol are two protocols to avoid deadlock. Deadlock is a safety property in which multiple processes are each waiting for the each other to finish before they continue. These properties are formally modelled in Groote and Willemse [2005] using a classical dynamic logic which allows them to quantify over paths. By extending the type system to dynamic logic, it would be possible to exploit the Curry-Howard correspondence to ensure these properties at compilation.

Wadge [1981] introduces the cycle sum test to avoid deadlock. Every node of a dataflow graph is associated with an integer. When a new stream is created, it is given with an new integer as a function of integers of streams that it depends on, as well as which operators are used to join these streams. If this new number is positive, the stream will not be waiting for its own output so will be deadlock-free. This enforces enough structure that streams can depend on loops while ensuring well-foundedness. This is a property that could be expressed with indices. However, the Jackrabbit type system is more strict and already rejects dead-locking programs, since programs cannot contain variables from the past. For certain problems, it would be useful to relax the next-step modality condition that a context be dropped at every time-step, and to instead use indices inspired by the cycle sum test.

The examples with bound streams expanded the range of the ratio between $A$s and $B$s. This range of allowable positions to see a certain type can also be modified in more complex ways. Given a polynomial $p$, it's possible to limit occurrences of $B$s to steps where the time since the beginning is $p(x)$ for $x \in \mathbb{N}$. By extending the index language to lists of booleans, $B$s can be limited to steps where the index list is true at position $|B| - |A|$. Boolean lists can be treated as binary numbers, and modified as numbers in a a sequence when passed to the next step.

Jackrabbit does not allow computation in types. By adding natural number addition to the index language, we can bound the number of $a$'s by more complex functions of $b$. For example, $|a| \leq |b|^2$ is shown in Figure 5–1. The number of $b$s seen is recorded with $m$, and the number of permissions $n$ exploits the fact that $m^2 - (m-1)^2 = m + (m-1)$.

# References

A. Abel. Termination checking with types. *ITA*, 38(4):277–319, 2004. doi: 10.1051/ita:2004015. URL `https://doi.org/10.1051/ita:2004015`.

B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985. doi: 10.1016/0020-0190(85)90056-0. URL `https://doi.org/10.1016/0020-0190(85)90056-0`.

D. Baelde and G. Nadathur. Combining deduction modulo and logics of fixed-point definitions. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, pages 105–114, 2012. doi: 10.1109/LICS.2012.22. URL `https://doi.org/10.1109/LICS.2012.22`.

G. Berry and L. Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In *Seminar on Concurrency, Carnegie-Mellon University, Pittsburg, PA, USA, July 9-11, 1984*, pages 389–448, 1984.

P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: A declarative language for programming synchronous systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 178–188, 1987.

A. Cave and B. Pientka. Programming with binders and indexed data-types. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 413–424, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. doi: 10.1145/2103656.2103705.

A. Cave, F. Ferreira, P. Panangaden, and B. Pientka. Fair reactive programming. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 361–372, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535881.

C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997.*, pages 263–273, 1997. doi: 10.1145/258948.258973. URL `http://doi.acm.org/10.1145/258948.258973`.

V. Goranko and A. Galton. Temporal logic. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. The Metaphysics Research Lab, winter 2015 edition, 2015.

J. F. Groote and T. A. Willemse. Model-checking processes with data. *Science of Computer Programming*, 56(3):251–273, 2005.

T. Hagino. A typed lambda calculus with categorical type constructors. In *Category Theory and Computer Science, Edinburgh, UK, September 7-9, 1987, Proceedings*, pages 140–157, 1987. doi: 10.1007/3-540-18508-9_24. URL `https://doi.org/10.1007/3-540-18508-9_24`.

R. Jacob-Rao, B. Pientka, and D. Thibodeau. Index-stratified types. In *3rd International Conference on Formal Structures for Computation and Deduction, FSCD*

*2018, July 9-12, 2018, Oxford, UK*, pages 19:1–19:17, 2018. doi: 10.4230/LIPIcs.FSCD.2018.19. URL `https://doi.org/10.4230/LIPIcs.FSCD.2018.19`.

A. Jeffrey. LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Proceedings of the sixth workshop on Programming Languages meets Program Verification, PLPV 2012, Philadelphia, PA, USA, January 24, 2012*, pages 49–60, 2012. doi: 10.1145/2103776.2103783.

A. Jeffrey. Causality for free!: parametricity implies causality for functional reactive programs. In *Proceedings of the 7th Workshop on Programming languages meets program verification, PLPV 2013, Rome, Italy, January 22, 2013*, pages 57–68, 2013. doi: 10.1145/2428116.2428127.

W. Jeltsch. Towards a common categorical semantics for linear-time temporal logic and functional reactive programming. *Electronic Notes in Theoretical Computer Science*, 286:229–242, 2012.

H. Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California at Los Angeles, 1968.

D. Kozen. Results on the propositional $\mu$-calculus. *Theoretical computer science*, 27 (3):333–354, 1983.

N. R. Krishnaswami. Higher-order functional reactive programming without space-time leaks. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 221–232, 2013. doi: 10.1145/2500365.2500588. URL `http://doi.acm.org/10.1145/2500365.2500588`.

N. R. Krishnaswami, N. Benton, and J. Hoffmann. Higher-order functional reactive programming in bounded space. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 45–58, 2012. doi: 10.1145/2103656.2103665. URL `http://doi.acm.org/10.1145/2103656.2103665`.

R. Matthes. *Extensions of system F by iteration and primitive recursion on monotone inductive types*. PhD thesis, Ludwig Maximilian University of Munich, Germany, 1999.

P. F. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, Ithaca, NY, USA, 1988. AAI8804634.

B. C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002. ISBN 0262162091, 9780262162098.

A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57, 1977. doi: 10.1109/SFCS.1977.32.

P. Schroeder-Heister. Rules of definitional reflection. In *Proceedings of the Eighth Annual Symposium on Logic in Computer Science LICS '93, Montreal, Canada, June 19-23, 1993*, pages 222–232, 1993. doi: 10.1109/LICS.1993.287585. URL `https://doi.org/10.1109/LICS.1993.287585`.

C. Schürmann and F. Pfenning. Automated theorem proving in a simple meta-logic for LF. In *Automated Deduction - CADE-15, 15th International Conference on Automated Deduction, Lindau, Germany, July 5-10, 1998, Proceedings*, pages 286–300, 1998. doi: 10.1007/BFb0054266. URL `https://doi.org/10.1007/BFb0054266`.

W. W. Wadge. An extensional treatment of dataflow deadlock. *Theoretical computer science*, 13(1):3–15, 1981.

H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pages 214–227, 1999. doi: 10.1145/292540.292560. URL `http://doi.acm.org/10.1145/292540.292560`.