Integrating runtime identity functions into a dependent type system

Master's Thesis

José Carlos Padilla Cancio

Integrating runtime identity functions into a dependent type system

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

José Carlos Padilla Cancio
born in Havana, Cuba

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Cover picture: Medieval rendering ghost.

Integrating runtime identity functions into a dependent type system

Author: José Carlos Padilla Cancio
Student id: 5224969

Abstract

Dependently typed languages allow developers to enforce compile time correctness of programs via the type system. These guarantees however, have to be proven with code, incurring a runtime and memory overhead. These costs can be avoided by using erasure (based on Quantitative Type Theory (QTT)) to omit code marked as erased (e.g. the aforementioned guarantees), which enables a separation between compile-time and run-time concerns.

Erasure annotations can give rise to types that are nominally different but structurally equal at runtime. We name functions between these types that behave like the identity at runtime, runtime identity (runid) functions. Current solutions do not have a structured way to reason about these runid functions as a first class member of the type system. This means programmers have no way to enforce that the compiler will erase these functions nor use the information of runid status to propagate optimizations, like defining runid functions that are polymorphic on some underlying runid function.

This thesis introduces a lightweight core language that extends a QTT-style, intensional Martin-Löf Type Theory (MLTT) with explicit markers for runid functions. We extend the type system with a static check that ensures runid-marked functions are equivalent to the identity function at run-time, using a novel run-time equivalence relation.

As a secondary contribution, we define a semantics for our language inspired by Normalization by Evaluation (NbE) [ACD07]. Our semantic domain is extensional, i.e. function equality is extensional, and agnostic to the compilation target, providing a clean model for reasoning about erased and runtime identity behaviour. We prove the soundness of our static analysis by showing that runid-equivalent terms are mapped to equal semantic values.

Thesis Committee:

Chair:	Dr. J.G.H. Cockx, Faculty EEMCS, TU Delft
Committee Member:	Dr. A. Panichella, Faculty EEMCS, TU Delft
University Supervisor:	B. Liesnikov, Faculty EEMCS, TU Delft

Preface

The cover picture of this thesis contains a ghost. The natural interpretation is that I am dealing with erasure, with things removed, with things gone. In some sense this is true. But this view is lacking, incomplete. Ghosts are not just gone; they are remnants of a past, influencing and determining the present. In many cultures, there is a notion of ghosts as haunting us, often presented and interpreted as some mythologized supernatural force. But the truth is that we do have ghosts, and they do haunt us. We are defined by that which comes before us, by those who walked the paths we now walk, by those who were there but now aren't.

In a way, types — and especially erased types — operate like a sort of ghost of computing. Invariants, contracts, promises made that disappear as we travel down the layers of abstraction, yet, despite being gone, are metaphysically present in some way. This thesis is about pulling more things (runtime identity functions) into the realm of ghosts, an admittedly macabre endeavour when worded this way. But in a sense, this essence of "ghostness" has nothing to do with being dead, but rather with influence maturing. As our code travels down to the metal, we no longer need to manually intervene, because our ghosts of previous assurances gently guide what comes out of each compilation step. Kind of like how, as a child matures, its parents slowly intervene less and less and become a "ghost": an influence that defines the potentiality of the child, rather than an active enforcer.

I've grown a lot since I landed at Schiphol five years ago. This thesis is the result of years of growing and learning. If it were not for the "ghosts" of my surroundings, I could not have reached the stage I am in now. So I would like to thank them.

To my supervisors Bohdan and Jesper: thank you for your invaluable guidance and support. Writing a thesis is a like navigating turbulent waters, and your support was like a lightpost in the storm.

To my dear friend William: thank you for recommending the functional programming course, fully unaware of the deep rabbit hole you were about to send me down, and thank you for the many laughs to help lighten the load.

To my friends Zoya, Matteo, and Puja: thank you for days of working together, spitballing ideas with each other, and working through our own theses in solidarity.

To my sister Isa: gracias por tu apoyo, sin el cual no hubiera logrado llegar a las alturas que he llegado.

To my parents: gracias por las décadas de amor y apoyo, criándome a la persona que soy hoy, y gracias por el privilegio de tener la vida que he tenido.

José Carlos Padilla Cancio
Delft, the Netherlands
September 2, 2025

Contents

List of Figures

List of Listings

Chapter 1

Introduction

Programmers use dependent type systems to express precise constraints on program behaviour. These constraints are specified within the type system itself, rather than relying on external specifications or verification tools. The compiler/type system then verifies that these constraints hold statically.

Dependent types can be interpreted via the Curry-Howard correspondence : types are logical statements, programs of these types are proofs of these statements. Put differently, in dependent type systems we can express properties as types and prove the property with the underlying program for the associated type.

Dependent types sit at the intersection between mathematics and computation, so we will give an intuitive introduction to their function from the perspective of the programmer: focusing on computation and how we define data types. From there we explain how this powerful programming style can incur costs and what methods we currently use to curtail these costs. Our contribution extends these methods to support runtime identity (runid) functions, i.e. functions which manipulate erased content but leave run-time relevant content untouched.

1.1 Dependently typed languages

The core property of dependent types is that types can *depend* on values. This leads to a more expressive type system than, for example, a polymorphic type system[1] where types can depend on types: such as lists $\forall A.\texttt{List}\ A$ that are generic on their carrier type A.

Let us take a closer look at an example of a dependent type by looking at how dependent function types work. In regular type systems, a function $f : A \to B$ will bind the argument $a : A$ in the body of f. However, dependent functions $f : (a : A) \to B$, also bind $a : A$ in the body of the return type B. Since a is in scope, B can be defined as a computation procedure that *depends* on the value of a — for example $f : (b : \mathbb{B}) \to (\texttt{if}\ b\ \texttt{then}\ X\ \texttt{else}\ Y)$ with $\mathbb{B}$ as the type of booleans and X, Y as arbitrary types. So $f(\texttt{true}) : X$ and $f(\texttt{false}) : Y$; the return type of the function is different for different inputs.

Full dependent type systems are used in a number of programming languages and projects. These range from pure theoretical settings of mechanizing mathematical proofs to verification of real world code such as compilers. Some examples are : "CompCert" [Ler+16] a formally verifed optimizing C[2] compiler, and "sel4" [Kle+09], a formally verified microkernel.

Some languages also implement partial dependent types, like Rust's "const generics" or C++ templates. In Rust, "const generics" allow types to depend on some compile-time constants (but not run-time values)— arrays being the canonical example: `[T;N]` is an array with

[1] also known as generics

[2] large subset of C99

entries of type `T` and `N` is a compile-time constant indicating the size of the array. This enables Rust to, for example, perform compile-time bounds checking for constant index values [3].

1.1.1 Inductive families

We can add dependency to data types as well! The paradigmatic examples used to show how dependent data types extend non-dependent ones are lists and vectors. Listing 1 introduces a definition of both in Agda.

```
data List (A : Set) : Set where
  []  : List A
  _::_ : (x : A) → (xs : List A) → List A

data Vec (A : Set) : (n : ℕ) → Set where
  []  : Vec A zero
  _::_ : ∀ {n} (x : A) → (xs : Vec A n) → Vec A (suc n)
```

Listing 1: Definition of lists and vectors in Agda

In Agda, the `data` keyword declares a new inductive type with parameters before the colon and indices after. Therefore, `data List (A : Set) : Set` declares a type `List` polymorphic on some type `A`. Lists are defined with two constructors: one for the empty list `[] : List A` and cons for non-empty lists `_::_ : (x : A) (xs : List A) → List A`. Cons takes an element (head) and a sublist (tail) and prepends the head to the tail. `Lists` are the Functional Programming (FP) implementation of linked lists. The underscore character in a function is used for "mixfix" notation, so the arguments to the cons function are supplied like this: `a :: as`.

On the other hand `data Vec (A : Set) : (n : ℕ) → Set` declares `Vec` which is parametric on `A` but indexed by `n`. Vectors are lists *indexed* by their length. The index argument $n : \mathbb{N}$ in the type signature shows us this dependence. Empty vectors have index 0. Non-empty vectors are constructed using the vector cons constructor `_::_ : ∀ {n} → (x : A) → (xs : Vec A n) → Vec A (suc n)`. The curly braces `{n}` is for implicit arguments. Arguments marked implicit can be omitted and Agda will infer their value, if it is able to. As such, visually the constructor looks the same `a :: as`, however, this is because Agda can infer the value for n. Under the hood, the explicit function call is `_::_ {n} a as`[4].

The type signature for the vector cons almost lines up with that of lists: we take a head and append a tail. The distinction lies in the extra index argument n, which shows us that prepending an element a to a list as of length n produces a list of size $1 + n = \texttt{suc}\ n$. Our constructor needs this index argument; without it we could specify neither the type of the tail `xs : Vec A ?`, nor the result term `Vec A (suc ?)`.

Vectors also tie back to our previous example of Rust arrays; encoding length information directly into the type allows us to perform length-sensitive operations (like indexing) on vectors without having to worry about bounds errors. For example, a safe and total[5] indexing function would have the type `index : {n : ℕ} → (i : ℕ) → Vector A n → i < n → A`. This type signature contains the guarantee $i < n$ that the index is within the bounds of the length. We know we will never perform an out-of-bounds index $i \geqslant n$, because we would have to provide evidence that it is in bounds $i < n$, which would contradict $i \geqslant n$.

`Vec` here is an example of an inductive family [Dyb94]. Inductive families generalize inductive data-types, which programmers may know (lists, trees, maps, tuples...), to depen-

[3] https://doc.rust-lang.org/reference/expressions/array-expr.html
[4] implicit arguments are excluded from mixfix notation
[5] Will never crash and is defined on all its inputs

dent types. Other data types in dependently typed languages include co-inductive, inductive-inductive, inductive-recursive and higher-inductive types. Inductive types, however, encode the common use case of data in programming: Finite[6], recursively structured, data.

1.2 Erasure

We have shown that dependent types enable strong correctness guarantees by allowing programs to express and enforce properties within their type signatures. In doing so, they introduce a natural and intuitive distinction between *computation* and *assurance*. Some parts of a program are relevant to run-time behaviour, while others[7] exist solely for compile-time verification. Ideally, we want to separate these concerns: run-time-relevant code should remain in the compiled program, while compile-time-only components can be checked statically and then safely erased.

This distinction is clearly illustrated by the vector example discussed earlier. The constructor `_::_` carries a length index `n`. For instance, `_::_ {2} a (_::_ {1} b (_::_ {0} c []))` encodes a three-element vector with explicitly stated lengths. Clearly, the information being stored is redundant. While these indices are essential for static analysis and type checking, retaining all 100 successive indices in a 100-element list at run-time would be wasteful, especially if such data is never used during execution.

Run-time-irrelevant indices can introduce overhead in both memory usage and computation time, although the extent of this overhead depends on the evaluation strategy of the language. In lazy languages, where function arguments are only evaluated when needed, run-time costs are lower but still present. The primary costs are in memory utilization but there are still some computation costs incurred [Tej20].

To this end dependently typed languages have come up with different mechanisms to handle run-time irrelevant data. Languages like Rocq[8] separate the language into two worlds, one for run-time code and one of compile time code. Code belonging to the compile time section of the language is not compiled. These two are not allowed to intermingle which gives a limited degree of granularity.

Languages with erasure based on QTT [Atk18], like Agda [Nor08], allow us to mix run-time and compile time code, as long as run-time code does not depend on compile time code. For example in Listing 2 shows how we define a vector with erased length indices as erased. We do this by marking the argument to the type (index) and the argument to the cons constructor as erased with the annotation `@0`. Below the Agda code we show what it roughly compiles to: the corresponding Haskell data type has no arguments for length as they have been marked erased. The constructors are: the empty list, which takes to arguments, and the cons constructor, which takes a head and a tail argument.

```
data Vec (A : Set) : @0 ℕ → Set where
  []  : Vec A zero
  _::_ : ∀ {@0 n}  → (x : A)  → (xs : Vec A n) → Vec A (suc n)

-----------------------------------------------

data Vec a = nil | cons a (Vec a)
```

Listing 2: Erased vector in Agda and its compilation to Haskell

[6] inductive types do not include infinitely recursive data such as streams. Such data is covered by co-inductive types

[7] such as proofs or type-level indices

[8] formerly known as Coq

1.3 Run-time identity functions

Erasure gives rise to different views [Wad87; MM04] on the same data . For example: after erasure, vectors and lists have the same structure and contents. They represent the same run-time data in the same structure, but with different erased assurances.

Since each view carries different properties, we sometimes need to map between two views, just to get the proofs we need. For example, the function `listToVec` defined in Listing 3 maps from a list to a vector. We reconstruct the list argument by pattern matching. In the cons case we insert the length (using a length function).

```
listToVec : (l : List A) -> Vec A (length l)
listToVec [] = []
listToVec a :: as = _::_ {length as} a (listToVec as)
```

Listing 3: listToVec function in Agda

The run-time-relevant data is not changed after executing the function. We call functions that behave like this: "run-time identity" (runid) functions. We want our compiler to recognize these functions and erase them (or replace standalone instances with an identity function `f a = a`). However existing compilers will compile this function naively as in Listing 4. Additionally it would be nice to be able to have a type level assurance that this will be optimized away.

```
listToVec :: List a -> Vec a
listToVec [] = []
listToVec (cons a as) = cons a as
```

Listing 4: listToVec compiled to Haskell

Standard QTT does not have any mechanism for annotating or verifying these functions. Luckily we already have the required building blocks: we have a mechanism for marking data and computation as erased; we can analyse function behaviour after erasure and determine its runid status.

There is some limited work in this field. Most of this work is based on ornaments [McB10]—which give a mechanism for reasoning about data types as extensions of other data types in an explicit manner. Unfortunately, approaches based on this framework have some costs.

Chiefly, they tend to rely on unergonomic encodings of data types. Programmers have to manually write mappings from existing data types into ornaments. These mappings also tend to imply manual proving of certain conditions that are implicit in the data type. Vectors would be encoded as $\texttt{Vec}\,A\,n = \{l : \texttt{List}\,A \mid \texttt{length}(l) = n\}$: The programmer has to manually separate lists from their length and prove that the length will align with the index. We would much rather have a "plug and play" solution that works on existing data types with minimal work.

1.4 The goal

Our approach is an inversion of ornament-based approaches. They focus on the internal representation of data types, while we focus on an analysis of existing types. We imagine a type system extended with annotations for saying: types $A \sim B$ are the same at run-time and $f : A \to B$ is a runid function.

Listing 5 shows a mock example of how these annotations could look in Agda. We state that the vector type should be considered equivalent to lists and tell the type system which constructors should be paired up together (trivial for now). We can check that this mapping

from lists to vectors is correct by comparing the function signatures of the paired up constructors and see that they are equivalent at run-time. From there we mark the `listToVec` function as runid, which the type system can verify because it now considers list and vector constructors as equivalent.

The annotations give enough information for checking these conditions without user-provided code. The programmer does not need to rewrite any data types, nor give any manual proofs.

These annotations would be integrated into the type system, enabling the programmer to use this type information as any other type. For example, they could define runid higher-order functions— which in turn empowers the programmer to define generic functions that leverage runid status, such as a generic runid map function on lists :
`@runid map : (@runid f : A -> B) -> List A -> List B.`

The type system could even infer this runid status, possibly even ad-hoc, which would reduce the number of user-provided annotations. Additionally, inference would allow the programmer to 'lift' regular higher-order functions to runid status. Returning to maps on lists: `map : (f : A -> B) -> List A -> List B` would not be runid but
`map g : @runid List A -> List B` would be, assuming that `g` is runid.

```
data @repr=List Vec (A : Set) : @0 ℕ → Set where
    @constructor=[]
    []  : Vec A zero
    @constructor=_::_
    _::_ : ∀ {@0 n} (x : A) (xs : Vec A n) → Vec A (suc n)

@runid listToVec : (l : List A) -> Vec A (length l)
listToVec [] = []
listToVec a :: as = a :: (listToVec as)
```

Listing 5: Mock data type and runid annotations

1.5 Contributions

The problem context gives us certain goals. We wish to work with the existing type system as opposed to giving a new underlying theoretical basis. We want to see what results we can obtain with no new user-provided code. Thus our research question is: "How could we design a type system for runid functions based on exhaustive annotations but no extra user-provided code". We assume that the runid functions are already present and do not count as extra user-provided code. Given the novel nature of this research we chose to emphasize exploring the theoretical underpinnings of such a type system.

Our contributions to address this research question are as follows:

- Runid Type Theory (RTT): A core type theory with exhaustive runid and erasure annotations, partially formalized in Agda. The type theory enforces erasure with a simplified version of QTT. Runid terms are typed by way of a syntactic run-time-equivalence relation $\Gamma \vdash a \sim_r b$. The relation denotes that two terms will compile to the same value.
- We give a mechanism of how to implement this as a language feature.
- A denotational semantics of run-time equivalence post-erasure using an extensional semantic domain based on the PER model from NbE [Abe13]. Terms $\Gamma \vdash a : A$ are mapped to functions on environments where the result type is the semantic type for A.

- A proof of soundness of (a variant of) our syntactic run-time equivalence relation with regard to this semantics in the style of logical relations. We show that terms related by our run-time equivalence relation are mapped to equal values in our semantic domain, after erasure.

Chapter 2

BTT: Our Quantitative Type Theory

We begin by introducing the base of our prototype Base Type Theory (BTT) which is a simplified QTT [Atk18]. Inspired by Agda, our type theory is an intensional dependent type theory with predicative universe levels. Variables are formalized using de Bruijn indices but will be presented using named variables for convenience in this thesis. Since the variable names are only for show, we employ the Barendrengt convention and assume that all variable names are fresh and there is no name capture. We have partially formalized BTT in Agda [1].

First we explain the basics of how to read and interpret a type theory, then give the proper definition of BTT: We begin by introducing the syntax. Then explain the usage modality and its operations. From there we define the typing judgements, first focusing on the rules for type formers and then on the rules for terms. In the end we explain the conversion rules that give rise to our definitional equality relation.

2.1 How to read a type theory

Before we go on to define our type theories it is instructive to explain what a type theory is, or more importantly, how it is specified. We will give a short introduction by way of a simple language with two values: natural numbers and functions.

We can define the syntax of terms and types with a grammar. The grammar is given in Backus-Naur form: the left hand side of $::=$ indicates the variables used to indicate the current class of strings, the right-hand side indicates the terminal and recursive strings, separated by $|$. For example `suc` a is a recursive term string with some term string a.

$x, y, z \in$ `String`

$A, B ::= (x : A) \to B$		Dependent functions
	$\mid$ `Nat`	Base types
	$\mid$ `Set`	Universes

Figure 2.1: Example syntax for types

A type theory gives us a mechanism to assign a type A to a term a: $a : A$. Each lambda constructor binds a variable which we need to keep track in a context $\Gamma ::= \varnothing \mid \Gamma, x : A$ which assigns a type to each variable. Putting this together we can define typing judgements $\Gamma \vdash a : A$ which can be read as: term a has type A in (typing) context Γ.

[1] https://github.com/J0s3c4rl0s/run-time_id_fn

$$
\begin{array}{rll}
 & x, y, z : \mathit{String} & \\
a, b, c, d, n, P ::= & x & \text{Variable} \\
 & \mid \lambda(x : A).b \mid a \cdot c & \text{Functions} \\
 & \mid \texttt{suc}\ a \mid \texttt{z} \mid \texttt{elNat}\, a\, P\, b\, c & \text{Nats}
\end{array}
$$

Figure 2.2: Example syntax for terms

A type theory gives a set of logical rules for how to derive these judgements from previous conditions. These rules are typically indicated in inference diagrams, taken from formal logic. Figure 2.3 gives a simple example of such a diagram. The bottom half P is the consequence and holds if the above conditions A, B also hold.

$$\frac{A \quad B}{P}$$

Figure 2.3: Inference diagram

Equipped with this knowledge of how to read inference diagrams we can specify the typing rules for our language. In general there are three categories of rules for typing (or syntactic terms):

1. Formation rules tell us how to "form" a type: e.g.

$$\frac{\Gamma \vdash A : \texttt{Set} \quad \Gamma, x : A \vdash B : \texttt{Set}}{\Gamma \vdash (x : A) \to B : \texttt{Set}}$$

2. Introduction rules tell us how to "introduce" a term of a type, which give us the rules for the constructor: e.g.

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda(x : A).b : (x : A) \to B}$$

3. Elimination rules tell us what are well defined computations, which correspond to eliminators: e.g.

$$\frac{\Gamma \vdash f : (x : A) \to B \quad \Gamma \vdash a : A}{\Gamma \vdash f \cdot a : B}$$

We exclude universe levels from the text since universe levels do not interact with our main results. Universe levels $\texttt{Set}_n : \texttt{Set}_{n+1}$ are a mechanism used to maintain logical soundness by avoiding Girard's paradox $\texttt{Set} : \texttt{Set}$.[2]

2.2 Syntax

Our syntax is defined with "usage annotations" π, σ, ρ, contexts Γ, types A, B, C and terms a, b, c. We do not distinguish between types and terms in our formalization, but we will use the custom of writing types in uppercase and terms in lowercase, with the exception of type functions, i.e. terms that return types (often denoted P).

Figure 2.4 defines our syntax for contexts. Contexts are maps from variable names to their type and usage.

[2]The paradox is the type-theoretic equivalent to Russell's paradox in (naive) set theory (Does the set of all sets that do not contain themselves, contain itself?). Universe levels resolve this by having a hierarchy of universes so `Set i : Set (i + 1)`, no universe contains itself and every universe is a member of a type.

$$\Gamma ::= \varnothing \quad | \quad \Gamma, x \overset{\sigma}{:} A$$

Figure 2.4: Syntax for Contexts

The syntax for type formers is defined in Figure 2.5. Note that as we have a dependent type theory some types can depend on terms. Most famously $\texttt{Vec}\ A\ n^0$ where n is some term indicating vector length. The usage modality in types are annotated by their usage with two forms of annotations: explicit and position-based. Annotations on explicit bindings $x \overset{\sigma}{:} A$ are relatively straightforward and found in dependent types (pairs and function spaces).

Position-based annotations such as $A\ ^{\sigma}\uplus^{\rho} B$ inform us of the usage that will be bound to the relevant constructor arguments. This second case derives from the fact that, in languages like Agda, type parameters and indices would be expressed as bindings e.g. `_+_ : (@0 A) -> B -> Set` for sum. We do not support general inductive types nor allow partial application to the type formers. Because of this, we annotate the arguments directly.

$$x, y, z, p \in \texttt{String}$$

$$\begin{array}{lll}
A, B, P ::= (x \overset{\sigma}{:} A) \to B & & \text{Dependent functions} \\
\quad | \quad \texttt{List}\ A \quad | \quad \texttt{Vec}\ A\ n^{\sigma} & & \text{Recursive types} \\
\quad | \quad A\ ^{\sigma}\uplus^{\rho} B \quad | \quad (x \overset{\sigma}{:} A) \times B^{\rho} & & \text{Sum and product type} \\
\quad | \quad \texttt{Nat} & & \text{Nats} \\
\quad | \quad \texttt{Set} & & \text{Universes}
\end{array}$$

Figure 2.5: Syntax for types

Terms are introduced with the rules in Figure 2.6: we present per type in an alternating fashion its constructor(s) and then its eliminator.

Regarding the constructors note that analogously to types we annotate bindings (e.g. for abstractions) and constructors (e.g for left injection of sums) with relevant usages. Our formalization annotates type parameters and indices in eliminators, but for readability we omit this from the syntax here as well.

$$\begin{array}{lll}
a, b, c, d, n, P ::= x & & \text{Variable} \\
\quad | \ \lambda(x \overset{\pi}{:} A).b \mid a \overset{\pi}{\cdot} c & & \text{Functions} \\
\quad | \ ^{\pi}(a, b)^{\rho} \mid \texttt{el}\ ^{\pi}\times^{\rho}\ a\ P\ b & & \text{Products} \\
\quad | \ \texttt{suc}\ \ a \mid \texttt{z} \mid \texttt{elNat}\ a\ P\ b\ c & & \text{Nats} \\
\quad | \ \texttt{cons}_l\ a\ b \mid \texttt{[]}_l \mid \texttt{elList}\ a\ A\ P\ b\ c & & \text{Lists} \\
\quad | \ \texttt{cons}_v^{\pi}\ a\ n\ b \mid \texttt{[]}_v^{\pi} \mid \texttt{elVec}^{\pi}\ a\ P\ c\ d & & \text{Vecs} \\
\quad | \ \texttt{inl}^{\pi,\rho} a \mid \texttt{inr}^{\pi,\rho} b \mid \texttt{el}\ ^{\pi}\uplus^{\rho}\ a\ P\ c\ d & & \text{Sums}
\end{array}$$

Figure 2.6: Syntax for terms

2.3 Quantities

Based on QTT [Atk18, section 2], every variable binding contains a quantity (also called usage). The erased usage 0 denotes variables that are not present at run-time, and the present usage ω variables or terms that are present at run-time.

Atkey defines these as a semi-ring and defines operations for argument sharing and enforcing the separation of run- and compile-time. However we do not need this entire machinery, it suffices to know that $0 < \omega$ and have a multiplication operation $\sigma\pi$. Multiplication can be thought of in this context as selecting a quantity, more specifically collapsing the usage to $0 = 0\sigma = \sigma 0$ when either argument is 0.

2.4 Typing rules

We now present the typing rules on our syntax to give a static semantics to what a well-formed expression is. We define the judgement $\Gamma \vdash a \overset{\sigma}{:} A$ to mean that term a has type A in context Γ in mode σ. Following [Atk18, section 2.1.3] the "mode" σ splits our type theory in two halves: erased 0 and present ω; where erased segments are relevant to the compile-time and present segments are relevant to the run-time.

For [Atk18] these modes differ from just the quantities as he accepts a general semiring for usages. However as our quantities are just erased and present we can neatly reuse them for the different modes of our typing derivations.

Substitution is defined as $a[x \mapsto b]$ where each instance of x in b is substituted for a.

We will implicitly assume that — when dealing with pairs or sums — at least one of the usage annotations is in run-time position. Otherwise, we could accept terms in run-time positions that are empty and thus non-sensical.

2.4.1 Types

Type formers are always typed in erased position $\Gamma \vdash A \overset{0}{:} \mathtt{Set}$, as type formation is only relevant to the compile time. Apart from this specific modification, the typing rules adhere to standard formulations commonly found in the literature.

Our dependent pairs are different to those given by [Atk18] (which they call dependent tensor) because they only support a usage annotation on the LHS of the product while we support usage annotation on both sides.

$$\frac{\Gamma \vdash A \overset{0}{:} \mathtt{Set} \quad \Gamma, x \overset{\pi}{:} A \vdash B \overset{0}{:} \mathtt{Set}}{\Gamma \vdash (x \overset{\pi}{:} A) \to B \overset{0}{:} \mathtt{Set}} \vdash \Pi \qquad \frac{\Gamma \vdash A \overset{0}{:} \mathtt{Set} \quad \Gamma, x \overset{\pi}{:} A \vdash B \overset{0}{:} \mathtt{Set}}{\Gamma \vdash (x \overset{\pi}{:} A) \times B^{\rho} \overset{0}{:} \mathtt{Set}} \vdash \Sigma$$

$$\frac{\Gamma \vdash A \overset{0}{:} \mathtt{Set}}{\Gamma \vdash \mathtt{List}\ A \overset{0}{:} \mathtt{Set}} \vdash \mathtt{List} \qquad \frac{\Gamma \vdash A \overset{0}{:} \mathtt{Set} \quad \Gamma \vdash n \overset{\sigma}{:} \mathbb{N}}{\Gamma \vdash \mathtt{Vec}\, A\, n^{\sigma} \overset{0}{:} \mathtt{Set}} \vdash \mathtt{Vec}$$

$$\frac{}{\Gamma \vdash \mathtt{Nat} \overset{0}{:} \mathtt{Set}} \vdash \mathbb{N} \qquad \frac{}{\Gamma \vdash A\, {}^{\sigma}\!\uplus^{\rho} B \overset{0}{:} \mathtt{Set}} \vdash \uplus$$

$$\frac{\Gamma \vdash a \overset{0}{:} A \quad \Gamma \vdash b \overset{0}{:} A}{\Gamma \vdash a \cong b \overset{0}{:} \mathtt{Set}} \vdash \cong$$

Figure 2.7: Type rules for type formers

2.4.2 Terms

Terms of types come in three kinds: variables, constructors and eliminators.

The variable rule checks that the context contains an entry with the same type and a usage greater or equal to the needed usage for the variable. This enforces the invariant that erased variables are not used in run-time position. Note that run-time variables can be used in erased position, we only restrict flow in one direction.

$$\frac{x \overset{\sigma'}{:} A \in \Gamma \quad \sigma \leqslant \sigma'}{\Gamma \vdash x \overset{\sigma}{:} A} \vdash var$$

Figure 2.8: Typing rule for variables

Constructors are well-typed when their arguments are well typed (e.g. $\Gamma \vdash h : A$ and $\Gamma \vdash t : \texttt{List}\ A$ for $\Gamma \vdash \texttt{cons}_l\, h\, t : \texttt{List}\ A$). For constructors of types that do not carry any usage annotations the mode σ is maintained. Constructors of types that do contain usage annotations multiply the mode by the usage when checking annotated terms, e.g. when checking the left value a in ${}^{\pi}(a, b)^{\rho}$ we scale the checking by π. As stated earlier this has the effect of shifting the typing mode to erased position if the value we are checking is introduced in erased position.

$$\frac{\Gamma, x \overset{\pi}{:} A \vdash b \overset{\sigma}{:} B \quad \Gamma \vdash A \overset{0}{:} \texttt{Set}}{\Gamma \vdash \lambda(x \overset{\pi}{:} A).b \overset{\sigma}{:} (x \overset{\pi}{:} A) \to B} \vdash \lambda \qquad \frac{\Gamma \vdash a \overset{\sigma\pi}{:} A \quad \Gamma \vdash b \overset{\sigma\rho}{:} B[x \mapsto a]}{\Gamma \vdash {}^{\pi}(a, b)^{\rho} \overset{\sigma}{:} (x \overset{\pi}{:} A) \times B^{\rho}} \vdash \texttt{pair}$$

$$\frac{}{\Gamma \vdash \texttt{z} \overset{\sigma}{:} \texttt{Nat}} \vdash \texttt{z} \qquad \frac{\Gamma \vdash n \overset{\sigma}{:} \texttt{Nat}}{\Gamma \vdash \texttt{suc}\ n \overset{\sigma}{:} \texttt{Nat}} \vdash \texttt{suc}$$

$$\frac{}{\Gamma \vdash []_l \overset{\sigma}{:} \texttt{List}\ A} \vdash []_l \qquad \frac{\Gamma \vdash a \overset{\sigma}{:} A \quad \Gamma \vdash as \overset{\sigma}{:} \texttt{List}\ A}{\Gamma \vdash \texttt{cons}_l\, a\, as \overset{\sigma}{:} \texttt{List}\ A} \vdash \texttt{cons}_l$$

$$\frac{}{\Gamma \vdash []_v^{\pi} \overset{\sigma}{:} \texttt{Vec}\, A\, z^{\pi}} \vdash []_v \qquad \frac{\Gamma \vdash h \overset{\sigma}{:} A \quad \Gamma \vdash t \overset{\sigma}{:} \texttt{Vec}\, A\, n^{\pi} \quad \Gamma \vdash n \overset{\sigma\pi}{:} \texttt{Nat}}{\Gamma \vdash \texttt{cons}_v^{\pi}\, h\, (\texttt{suc}\ n)\, t \overset{\sigma}{:} \texttt{Vec}\, A\, (\texttt{suc}\ n)^{\pi}} \vdash \texttt{cons}_v$$

$$\frac{\Gamma \vdash a \overset{\sigma\pi}{:} A}{\Gamma \vdash \texttt{inl}^{\pi,\rho} a \overset{\sigma}{:} A\, {}^{\sigma}\uplus^{\rho} B} \vdash \texttt{inl} \qquad \frac{\Gamma \vdash b \overset{\sigma\rho}{:} B}{\Gamma \vdash \texttt{inr}^{\pi,\rho} b \overset{\sigma}{:} A\, {}^{\sigma}\uplus^{\rho} B} \vdash \texttt{inr}$$

Figure 2.9: Typing rules for constructors

Eliminators give rise to eponymous elimination rules and encode a computation procedure, such that feeding a constructor to an eliminator gives rise to a concrete computation (or reduction). Eliminators on data types are equivalent to structural induction/recursion on the data type.

Let us look at the typing of the eliminator on natural numbers $\texttt{elNat}\, n\, P\, b\, c$ to elucidate the typing rules on (recursive) data type eliminators. The eliminator is typed in mode σ and takes four arguments:

1. The scrutinee n, the argument being matched on, is type checked in the same position σ.

$$\frac{\Gamma \vdash n \overset{\sigma}{:} \texttt{Nat} \quad \Gamma \vdash P \overset{0}{:} (x \overset{\sigma}{:} \texttt{Nat}) \to \texttt{Set} \quad \Gamma \vdash b_z \overset{\sigma}{:} P \overset{\sigma}{\cdot} \texttt{z} \quad \Gamma, m \overset{\sigma}{:} \texttt{Nat}, p \overset{\sigma}{:} P \overset{\sigma}{\cdot} m \vdash b_s \overset{\sigma}{:} P \overset{\sigma}{\cdot} (\texttt{suc}\ m)}{\Gamma \vdash \texttt{elNat}\, n\, P\, b_z\, b_s \overset{\sigma}{:} P \overset{\sigma}{\cdot} n} \vdash \texttt{elNat}$$

Figure 2.10: Typing `Nat` eliminator

2. The motive P, which gives the return type per input, which is typed in erased position as it is a type function.

3. The branches b, c give the relevant execution code for the scrutinee. In cases where there are multiple constructors there is one branch per constructor (b is for z, c is for $\texttt{suc}\ m$). Each branch exposes the relevant arguments to the constructor by extending the context. If the data type is inductive (which `Nat` is), we also encode an induction procedure during elimination: Let $n = \texttt{suc}\ m$, the eliminator is applied to the subterm(s) m and provided to the inductive branch c, by way of the argument p. [3]

For applications $f \overset{\pi}{\cdot} a$ the rule checks in position σ that f is indeed a function with argument of quantity π. From there it checks that the argument a has the correct argument type in the $\sigma\pi$ mode, meaning the argument is checked in erased mode if either: the function takes erased argument or the typing is already in erased position.

This scaling operation is also present when type checking sum branches: erasure in sums indicates if the left or right constructor are erased, e.g. $A\, {}^{0}\!\uplus^{\omega} B$ has an erased left constructor. As such each branch is scaled to match the usage of the constructor: if the left constructor is erased then its corresponding branch must also be in erased position.

$$\frac{\Gamma \vdash f \overset{\sigma}{:} (x \overset{\pi}{:} A) \to B \quad \Gamma \vdash a \overset{\sigma\pi}{:} A}{\Gamma \vdash f \overset{\pi}{\cdot} a \overset{\sigma}{:} B} \vdash \cdot$$

$$\frac{\Gamma \vdash s \overset{\sigma}{:} A\, {}^{\pi}\!\uplus^{\rho} B \quad \Gamma \vdash P \overset{0}{:} (x \overset{\sigma}{:} A\, {}^{\pi}\!\uplus^{\rho} B) \to \texttt{Set} \quad \Gamma, x \overset{\pi}{:} A \vdash b_L \overset{\sigma\pi}{:} P \overset{\sigma}{\cdot} (\texttt{inl}^{\pi,\rho} a) \quad \Gamma, x \overset{\rho}{:} B \vdash b_R \overset{\sigma\rho}{:} P \overset{\sigma}{\cdot} (\texttt{inr}^{\pi,\rho} b)}{\Gamma \vdash \texttt{el}\, {}^{\pi}\!\uplus^{\rho}\, s\, P\, b_L\, b_R \overset{\sigma}{:} P \overset{\sigma}{\cdot} s} \vdash \texttt{el}\uplus$$

Figure 2.11: Typing rules for eliminators with scaling

2.4.3 Mode zeroing rule

Rule $\vdash$ TM0 is exactly the same as in QTT, it "state[s] that we can take any term and produce its 'resource free' counterpart in the $\sigma = 0$ fragment"[Atk18]. Note that this rule only holds in one direction, an arbitrary compile time judgement obviously cannot be assumed to also hold as a run-time judgement.

$$\frac{\Gamma \vdash a \overset{\sigma}{:} B}{\Gamma \vdash a \overset{0}{:} B} \vdash \texttt{TM0}$$

Figure 2.12: Typing rule to shift type checking mode.

[3] Traditionally the branches would be functions taking these extensions as arguments. By extending the context we increase syntactic readability as we write c instead of $\lambda(m \overset{\sigma}{:} \texttt{Nat}).\lambda(p \overset{\sigma}{:} P \overset{\sigma}{\cdot} m).c$

2.5 Equality

Since we are dealing with a dependently typed language, our types can depend on terms which themselves may describe some computation procedure. As a trivial example consider $\Gamma \vdash \mathtt{z} \overset{\sigma}{:} \mathtt{id} \overset{0}{\cdot} \mathtt{Nat}$, we can only type this correctly if we can show that $\mathtt{id} \overset{0}{\cdot} \mathtt{Nat}$ reduces to $\mathtt{Nat}$. As such we need some notion of computation-aware equality between types/terms. This is commonly achieved by defining a definition equality relation. For us this is an untyped conversion relation on terms $A \equiv B$, which we can use in our typing rules via the $\vdash \mathtt{conv}$ rule. This empowers us to resolve the previously mentioned example, as we should be able to reduce the function application to simply be equal to Nat.

$$\frac{\Gamma \vdash a \overset{\sigma}{:} A \quad A \equiv B}{\Gamma \vdash a \overset{\sigma}{:} B} \vdash \mathtt{conv}$$

Figure 2.13: Typing conversion rule

Our definitional equality contains the usual equivalence (reflexivity transitivity, symmetry), congruence and computation rules (β-reduction). Note that in the conversion relation the β reductions are modality agnostic, e.g. they will reduce $f \overset{0}{\cdot} a, f \overset{\omega}{\cdot} a, f \cdot_r a$ in the same manner. Congruence rules are only modality aware in that they will not mismatch modality annotations, e.g. $f \overset{\omega}{\cdot} a \not\equiv f \cdot_r a$.

2.5.1 Propositional equality

Sometimes we need to tell the type system that two terms are equal when it cannot tell just by definitional equality. In these situations we want to have a type to express equality which is fulfilled by propositional equality. Figure 2.14 showcases the typing rules for propositional equality. We only have one introduction for propositional equality $\mathtt{rfl}$ which which is the trivial reflexivity proof. When we eliminate on some equality we wish to check that the body type checks assuming that we rewrite the right term b for the left one a and that we have resolved the equality to $\mathtt{rfl}$.

$$\frac{\Gamma \vdash a \overset{\sigma}{:} A}{\Gamma \vdash \mathtt{rfl} \overset{\sigma}{:} a \cong a} \vdash \mathtt{rfl}$$

$$\frac{\begin{array}{l}\Gamma \vdash P \overset{0}{:} (x \overset{0}{:} A) \to (y \overset{0}{:} x \cong x) \to \mathtt{Set} \\ \Gamma \vdash b \overset{\sigma}{:} P \overset{\sigma}{\cdot} a \overset{0}{\cdot} \mathtt{rfl} \qquad\qquad \Gamma \vdash e \overset{\sigma}{:} a \cong b\end{array}}{\Gamma \vdash \mathtt{el} \cong^{\pi}\ e\, b\, P \overset{\sigma}{:} P \overset{\sigma}{\cdot} b \overset{0}{\cdot} e} \vdash \mathtt{el} \cong$$

Figure 2.14: Typing rules for propositional equality

2.6 Limitations

We only have a short overview of selected data types to explore the design of the type theory. Future work would expand this to general inductive families.

Chapter 3

RTT: adding runid functions to our language

In this chapter, we extend our type theory with the notion of runid functions. The philosophy underpinning this extension is that we think of our type theory as a core type theory where all runid terms are marked and thus do not have to be inferred or where runid inference is done in an earlier type-checker/compiler pass. These marked terms are then checked by our type system to verify that their run-time behaviour does in fact equate to that of an identity function, post erasure. This check is facilitated by our run-time equivalence relation $\Gamma \vdash a \sim_r b$ which states that in some context two terms will be equal post-compilation.

We are being intentionally vague for now about the definition of compilation or erasure procedure. We will define what we mean more rigorously when we define our semantics. For now, it suffices to have an intuitive understanding of compilation as an "optimizing" compiler which optimizes terms exclusively based on usage annotations and runid markers. Its behaviour will become more clear when we describe the individual rules.

Our typing rules and run-time analysis interdepend and take on different responsibilities. The run-time analysis verifies that terms will be equal post compilation, assuming annotations are valid. The typing rules on the other hand verify that annotations are correct assuming that our run-time analysis will reject terms which do not compile to the same term or value.

We will first present the syntax of the language with runid markers. Then we will describe the typing rules. Finally we will present the run-time relation which underpins our analysis.

3.1 Syntactic extension

We extend our syntax in Figure 3.1 by marking types and terms by a subscript $_r$ if they are in some sense runid.

1. Functions are the only type that can be runid, and must have non-erased inputs so we omit the usage annotation.
2. Lambdas constitute the only constructor that can be runid
3. The rest are eliminators on data types, following the pattern: $\texttt{el}_r\texttt{Type}$.

3.2 Typing runid terms

Since the runid markers are an extension of existing types, our typing rules follow a similar structure of first checking that the unmarked term is well typed and then checking the run-time behaviour of the term. This allows us to have a weak, but efficient, relation that depends

$$
\begin{array}{rll}
A, B, P ::= & (x : A) \to_r B & \text{Dependent functions} \\
a, b, c, d, n, P, ::= & & \\
\quad | & \lambda_r(x : A).B & \text{Abstraction} \\
\quad | & a \cdot_r b & \text{Application} \\
\quad | & \mathtt{el}_r{}^{\pi} \times^{\rho} a\, P\, b & \text{Product eliminator} \\
\quad | & \mathtt{el}_r\mathtt{Nat}\, a\, P\, b\, c & \text{Nat eliminators} \\
\quad | & \mathtt{el}_r\mathtt{List}\, a\, A\, P\, b\, c & \text{List eliminator} \\
\quad | & \mathtt{el}_r\mathtt{Vec}^{\pi}\, a\, P\, c\, d & \text{Vec eliminator} \\
\quad | & \mathtt{el}_r{}^{\pi} \uplus^{\rho} a\, P\, c\, d & \text{Sum eliminator}
\end{array}
$$

Figure 3.1: Runid extension

on typing and does not require us to check the validity of (runid) terms when expressing the relations between terms at run-time. We will define and explain the analysis underpinning this relation in the next section.

3.2.1 Functions

The function type and eliminator are the exceptions to this pattern of checking run-time equivalence when dealing with runid terms. We do not need to perform any analysis to accept runid type formation, merely well-formedness. Lambdas are accepted as runid under the requirement that their function body is equivalent to the argument to the function. Applications need no run-time equivalence analysis, they merely need to check that the provided function is typed as runid.

$$\frac{\Gamma \vdash A \overset{0}{:} \mathtt{Set} \quad \Gamma, x \overset{\omega}{:} A \vdash B \overset{0}{:} \mathtt{Set}}{\Gamma \vdash (x : A) \to_r B \overset{0}{:} \mathtt{Set}} \vdash \Pi_r$$

$$\frac{\Gamma \vdash \lambda(x \overset{\omega}{:} A).b \overset{\sigma}{:} (x \overset{\omega}{:} A) \to B \quad \Gamma, x \overset{\omega}{:} A \vdash b \ \sim_r \ x}{\Gamma \vdash \lambda_r(x : A).b \overset{\sigma}{:} (x : A) \to_r B} \vdash \lambda_r \quad \frac{\Gamma \vdash f \overset{\sigma}{:} (x : A) \to_r B \quad \Gamma \vdash a \overset{\sigma}{:} A}{\Gamma \vdash f \cdot_r a \overset{\sigma}{:} B} \vdash \cdot_r$$

Figure 3.2: Typing rules for runid function terms

3.2.2 Eliminators

To reduce notational noise, we omit the typing hypothesis that the unmarked version of the term is well-typed.

We only allow runid eliminators to be well-typed when the scrutinee is a variable a: we substitute into a the arbitrary constructor that matches the case — e.g. the empty list $b[a \mapsto \mathtt{[]}_l]$ and non-empty list $c[a \mapsto \mathtt{cons}_l\, h\, t]$ branches. We also substitute recursive subterms (like the tail of a list or the predecessor of a number) with the result of the recursive eliminator call. This models an induction procedure: by checking each branch, we ensure the condition holds in the base case (like $\mathtt{[]}_l$), and in the inductive cases (like $\mathtt{cons}_l\, h\, t$), we assume the subterms match the corresponding recursive results.

We would naively prefer to have a structured form of reasoning about keeping track of assumptions of run-time equivalence in our analysis: e.g. $\Gamma \vdash p \ \sim_r \ \mathtt{suc}\ m \implies \Gamma \vdash$

$b_s \sim_r \texttt{suc}\ m$. However including such hypotheses would violate strict positivity in the formalization and well-foundedness in the mathematical relation. As such we use substitution as a safe way to define this, but suggest future work to give a more structured approach to defining, say, an assumption context Δ for a more general system.

$$\frac{\Gamma_1, \Gamma_2 \vdash b_z[x \mapsto \texttt{z}] \sim_r \texttt{z} \qquad \Gamma_1, \Gamma_2, m \overset{\omega}{:} \texttt{Nat} \vdash b_s[p \mapsto m][x \mapsto \texttt{suc}\ m] \sim_r \texttt{suc}\ m}{\Gamma_1, x \overset{\omega}{:} \texttt{Nat}, \Gamma_2 \vdash \texttt{el}_r\texttt{Nat}\, x\, P\, b_z\, b_s \overset{\omega}{:} P \overset{\omega}{\cdot} x} \vdash \texttt{el}_r\texttt{Nat}$$

$$\frac{\Gamma_1, \Gamma_2 \vdash b_n[l \mapsto []_l] \sim_r []_l \qquad \Gamma_1, \Gamma_2, h \overset{\omega}{:} A, t \overset{\omega}{:} \texttt{List}\ A \vdash b_n[p \mapsto t][l \mapsto \texttt{cons}_l\, h\, t] \sim_r \texttt{cons}_l\, h\, t}{\Gamma_1, l \overset{\omega}{:} \texttt{List}\ A, \Gamma_2 \vdash \texttt{elList}\, l\, A\, P\, b_n\, b_c \overset{\omega}{:} P \overset{\omega}{\cdot} as} \vdash \texttt{el}_r\texttt{List}$$

Figure 3.3: Typing rules for `Nat` and `List` eliminators

Since erasure in sums implies erasure of constructors, we only perform analysis on the branches which are non-erased. It is nonsensical to relate terms in erased position, so we merely omit these comparisons. This induces 3 rules for each valid erasure configuration [1].

$$\frac{\Gamma_1, \Gamma_2, x \overset{\pi}{:} A, y \overset{\rho}{:} B \vdash b[c \mapsto {}^{\pi}(x, y)^{\rho}] \sim_r {}^{\pi}(x, y)^{\rho}}{\Gamma_1, c \overset{\omega}{:} (x \overset{\pi}{:} A) \times B^{\rho}, \Gamma_2 \vdash \texttt{el}_r\, {}^{\pi}\times^{\rho}\, c\, P\, b \overset{\omega}{:} P \overset{\omega}{\cdot} c} \vdash \texttt{el}_r \times$$

$$\frac{\Gamma_1, \Gamma_2, y \overset{\omega}{:} B \vdash b_R[s \mapsto \texttt{inr}^{0,\omega} y] \sim_r \texttt{inr}^{0,\omega} y}{\Gamma_1, s \overset{\omega}{:} A\, {}^{0}\uplus^{\omega} B, \Gamma_2 \vdash \texttt{el}_r\, {}^{0}\uplus^{\omega}\, s\, P\, b_L\, b_R \overset{\omega}{:} P \overset{\omega}{\cdot} s} \vdash \texttt{el}_r\, {}^{0}\uplus^{\omega}$$

Figure 3.4: Typing rules for runid products and sums

3.3 Run-time equivalence

Finally we come to the primary novel analysis of this thesis: the run-time equivalence relation $\Gamma \vdash a \sim_r b$ — which has so far been used in typing rules but not defined properly. The majority of the rules are not surprising: our run-time equivalence relation shares the same congruence and equivalence rules present in our conversion relation (but not the computation rules). The interesting rules concern terms annotated with erasure or marked with runid. In both cases terms are equated to their optimization.

3.3.1 Relating runid terms

Remember that we already perform a correctness check of any runid terms we encounter in our typing rules, so when we encounter them in our relation we assume they are correct. To this end, the runid functions correspond to identity functions, as shown in Figure 3.5. What remains are runid eliminators on data types, which all relate to their scrutinee: e.g. $\Gamma \vdash \texttt{el}_r\texttt{Nat}\, a\, P\, b\, c \sim_r a$ for some variable a.

[1] One could also have one rule with conditional checks on usage before requiring the analysis

$$\frac{}{\Gamma \vdash (a : A) \to_r B \ \sim_r \ (a : A) \to A} \sim\!\!\to_r$$

$$\frac{}{\Gamma \vdash \lambda_r(a : A).b \ \sim_r \ \lambda(a : A).b} \sim \lambda_r \qquad \frac{}{\Gamma \vdash f \cdot_r a \ \sim_r \ a} \sim \cdot_r$$

Figure 3.5: Relating runid functions

3.3.2 Relating types with erasure

The rules on types motivate the rules on terms. If two types are considered to be equivalent at run-time $\Gamma \vdash A \ \sim_r \ B$ then for each $\Gamma \vdash a : A$ then there should be some $\Gamma \vdash b : B$ $\Gamma \vdash a \ \sim_r \ b$. So we begin with types.

Figure 3.6 gives the rules for equating types with erasure information. Most types are related to their non-erased counterpart. The only caveat is that dependent function and pair types weaken the codomain of the result type with the argument variable $\Gamma, x \overset{0}{:} A \vdash B \overset{0}{:} \mathtt{Set}$. So we cannot just say $\Gamma \vdash (x \overset{0}{:} A) \to B \ \sim_r \ B$ since B is weakened with regards to Γ. Instead we pick a C that is strengthened with regards to x, i.e. C does not refer to x.

$$\frac{\Gamma, x \overset{0}{:} A \vdash B \ \sim_r \ C}{\Gamma \vdash (x \overset{0}{:} A) \to B \ \sim_r \ C} \sim_r (\overset{0}{:}) \to \qquad \frac{}{\Gamma \vdash \mathtt{Vec}\, A\, n^0 \ \sim_r \ \mathtt{List}\ A} \sim_r \mathtt{Vec}^{\,0}$$

$$\frac{\Gamma, x \overset{0}{:} A \vdash B \ \sim_r \ C}{\Gamma \vdash (x \overset{0}{:} A) \times B \ \sim_r \ C} \sim_r (\overset{0}{:}) \times \qquad \frac{}{\Gamma \vdash (x : A) \times B^0 \ \sim_r \ A} \sim_r (:) \times^0$$

$$\frac{}{\Gamma \vdash A \,{}^0\!\uplus B \ \sim_r \ B} \sim_r {}^0\!\uplus \qquad \frac{}{\Gamma \vdash A \uplus^0 B \ \sim_r \ A} \sim_r \uplus^0$$

Figure 3.6: Run-time equivalence rules on types

3.3.3 Relating terms with erasure

The rules on terms are motivated by the rules on types. As we said, if $\Gamma \vdash A \ \sim_r \ B$ then surely for any $\Gamma \vdash a : A$ there must be some $\Gamma \vdash b : A$ such that $\Gamma \vdash a \ \sim_r \ b$. We can find this b by way of the rules on terms. We present the rules for constructors and then for eliminators.

3.3.4 Constructors

Erased functions are related to a version of their bodies without reference to the erased argument (same procedure we used for function types). Right-erased pairs are related to their left term (and vice versa). A left-injection for a right-erased sum is just its content (and vice versa). Finally erased vectors are related to list constructors.

Note that we assume a correspondence between vectors and lists and thus a correspondence between their constructors. In a full system, correspondence of nominal types would not be hardcoded. Instead, you could obtain these rules by checking if relevant signatures are run-time equivalent.

The algorithm for including new rules would be: Compare the signatures of the type formers; if successful compare the signatures of the constructors, assuming that the nominal types are run-time equivalent. If this check succeeds you introduce rules equating the nominal type formers and the constructors. Let us apply this process to vectors and lists as an example. The vector signatures are on the LHS, the list signatures on the RHS.

$$\frac{\Gamma, x \overset{0}{:} A \vdash b \ \sim_r \ c}{\Gamma \vdash \lambda(x \overset{0}{:} A).b \ \sim_r \ c} \sim_r \lambda 0$$

$$\frac{}{\Gamma \vdash (a,b)^0 \ \sim_r \ a} \sim_r (,)^0 \qquad \frac{\Gamma, x \overset{0}{:} A \vdash b \ \sim_r \ c}{\Gamma \vdash {}^0(a,b) \ \sim_r \ c} \sim_r {}^0(,)$$

$$\frac{}{\Gamma \vdash \texttt{inl}^{,0} a \ \sim_r \ a} \sim_r \texttt{inl}^{,0} \qquad \frac{}{\Gamma \vdash \texttt{inr}^{0,} b \ \sim_r \ b} \sim_r \texttt{inr}^{0,}$$

$$\frac{}{\Gamma \vdash []^0_v \ \sim_r \ []_l} \sim_r []^0_v \qquad \frac{\Gamma \vdash a \ \sim_r \ b \quad \Gamma \vdash as \ \sim_r \ bs}{\Gamma \vdash \texttt{cons}^0_v \, a \, n \, as \ \sim_r \ \texttt{cons}_l \, b \, bs} \sim_r :: v_0$$

Figure 3.7: Constructor erasure rules

1. Compare the type formers' signatures:
   $\Gamma \vdash (A : \texttt{Set}) \to (n \overset{0}{:} \texttt{Nat}) \to \texttt{Set} \ \sim_r \ (A : \texttt{Set}) \to \texttt{Set}$. We now assume that $\Gamma \vdash \texttt{Vec} \, A \, n^0 \ \sim_r \ \texttt{List} \ A$ during our analysis of the constructors.

2. Compare the constructors:

   nil: $\Gamma \vdash \texttt{Vec} \, A \, \texttt{z}^0 \ \sim_r \ \texttt{List} \ A$. Holds by assumption.

   cons: $\Gamma \vdash \begin{array}{l}(n \overset{0}{:} \texttt{Nat}) \to \\ (x : A) \to \\ (xs : \texttt{Vec} \, A \, n^0) \to \\ \texttt{Vec} \, A \, \texttt{suc} \ n^0\end{array} \sim_r \begin{array}{l}(x : A) \to \\ (xs : \texttt{List} \ A) \to \\ \texttt{List} \ A\end{array}$

   The argument n gets erased. The equality holds by congruence on function types, via reflexivity $\Gamma \vdash A \ \sim_r \ A$ and by assumption $\Gamma \vdash \texttt{Vec} \, A \, \texttt{suc} \ n^0 \ \sim_r \ \texttt{List} \ A$.

Eliminators

Figure 3.8 showcases the inference rules for eliminators with partially erased types. Again, we are guided by the rules on types for eliminators. Function applications get equated to the function term, which is equated to the body when the function is resolved to lambda term. As for the datatype eliminators: If $\Gamma \vdash A \ \sim_r \ B$ then an eliminator for A is equated to an eliminator for B. Vector eliminators equal list eliminators, subject to strengthening of the branches. With dependent pairs the eliminators correspond to let bindings of the non-erased counterpart, where the body is the strengthening of the branch. Sums are defined similarly as let bindings of the inner value, where we select the non-erased branch for the body. Since we do not directly support let bindings in our language we simulate these with lambdas.

3.4 Limitations

Our rules are so far hardcoded for aligning nominally distinct but structurally equivalent types (i.e. lists and vectors). We give a procedure to determine when types should be considered structurally equivalent, however it would be nice for future work to define this more formally for general inductive families.

The solution of using substitutions to model induction hypotheses works but is not an elegant solution, it would be preferable for future work to explore how to generalize keeping track of assumed equivalences.

$$\frac{\Gamma \vdash f \ \sim_r \ f'}{\Gamma \vdash f \overset{0}{\cdot} a \ \sim_r \ f'} \sim_r \overset{0}{\cdot}$$

$$\frac{\begin{array}{c}\Gamma \vdash b_n \ \sim_r \ b'_n \\ \Gamma, n \overset{0}{:} \mathtt{Nat}, h : A, t : \mathtt{Vec}\, A\, n^0 \vdash b_c \ \sim_r \ b'_c\end{array}}{\Gamma \vdash \mathtt{elVec}^0\, a\, P\, b_n\, b_c \ \sim_r \ \mathtt{elList}\, a\, P\, b'_n\, b'_c} \sim_r \mathtt{elVec}^0$$

$$\frac{\Gamma, x \overset{0}{:} A, y : B \vdash b \ \sim_r \ c}{\Gamma \vdash \mathtt{el}^0 \times\, a\, P\, b \ \sim_r \ \lambda(y : B).c \cdot a} \sim_r \mathtt{el}^0 \times$$

$$\frac{\Gamma, x : B \vdash b_R \ \sim_r \ c}{\Gamma \vdash \mathtt{el}^0 \uplus\, a\, P\, b_L\, b_R \ \sim_r \ \lambda(x : B).c \cdot a} \sim_r \mathtt{el}^0 \uplus$$

Figure 3.8: Eliminator erasure rules

Chapter 4

Semantics

This chapter endeavours to give a semantics to the notion of "equivalent at run-time" that underpins our analysis of erasure and runid functions. The system thus far is very syntactical in nature: it concerns itself with literal syntax strings and manipulating them. We have not explored, however, what this analysis is supposed to *mean*, we have not given a semantics to our analysis.

We give a denotational semantics of equivalence at run-time by way of a partial equivalence relation (PER) model à la NbE [Abe13]. We use their notion of a fully normalized domain D which we map to for our comparison, but do not reify back into our original syntax, as we are only interested in equality in the domain.

We first introduce what a denotational semantics is and motivate it. Then explain the internal erasure operation. Then define the domain of values D that we map to. Then we define how to relate values as equivalent in our domain. Finally, we bring everything together to define a notion of semantically run-time equivalent terms which we use in our proof.

4.1 Denotational semantics

As mentioned earlier, our run-time equivalence is intended to capture the loose notion of two terms being "equivalent" at run-time post erasure. This analysis is intended to justify our runid compiler optimizations, as including them should not change the semantics of the resulting program.

Ultimately we are interested in runid functions which may have a different *syntax* but the same *behaviour* as an identity function. This implies we want some form of extensionality in our domain, we want functions that behave the same to be equal: $\forall a. f(a) = g(a) \implies f = g$.

Additionally, we may have multiple targets of compilation. Agda typically compiles to Haskell, but also has compilation backends for JavaScript and AGDA2HS [Coc+22] an alternative Haskell compiler. This complicates things somewhat. If we include runid functions then we are including an element into our type system that we have defined by way of the run-time, and by extension compilation (target). But if a language supports multiple compilers and targets (i.e. it is defined by specification not implementation), then we need a compiler agnostic semantics. Or at least a specification that is agnostic to implementation details.

This compiler generality motivates a denotational semantics. Denotational semantics give meaning to a program by way of some mathematical model. For the purposes of this thesis we would like to emphasize this as an abstraction of computation. We have a sense of computation and a sense of redundant or equivalent computation, but it does not matter what the specifics of this computation are, just what its result is.

Let us explain how this differs from "syntactic" approaches, using Turing machines as a friendly analogy for a syntactic approach. Turing machines abstract away the implementation details of hardware or software by giving a system of step-by-step operations (opera-

tional semantics). Denotational semantics abstract away the specifics of step-by-step computation by delegating that computation to some mathematical model. The simplest example of this is a language which only contains numbers being given the semantics of $\mathbb{N}$ as values and mapping functions to functions on natural numbers. Now that we are operating on mathematical functions we can, for example, state that two implementations of the Fibonacci function are the same because they give the same results for the same inputs, without concerning ourselves with their execution mechanism.

As such we define a domain D and an interpretation from our language to our semantic domain $[\![\_]\!]$. Alternative compilers respect the semantics of runid and erasure if they map to some language whose semantic model D' is isomorphic to our own, as illustrated in the following commutative diagram.

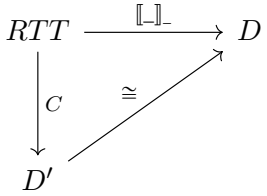

Figure 4.1: Compilation must respect our semantics up to isomorphism

4.2 Erasure function

As we said, our erasure semantics is defined by a partial erasure function $\downarrow \_ : \mathtt{RTT} \to \mathtt{RTT}$ from our core type theory to an erasure-free subset:

1. Terms marked erased are dropped, e.g. $\downarrow (a \overset{0}{:} A) \to B = \downarrow B$
2. runid marked terms have their marking dropped, e.g. $\downarrow f \cdot_r a = \downarrow f \cdot \downarrow a$
3. Type annotations and motives are ignored, e.g. $\downarrow \mathtt{elNat}\, a \_\, b\, c = \mathtt{elNat}\ \downarrow a \_\ \downarrow b\ \downarrow c$
4. The function is undefined for explicitly erased terms, e.g. $\downarrow x$ if $x \overset{0}{:} A \in \Gamma$ or $\downarrow \mathtt{inl}^{0,\omega} a$.

An exhaustive definition of the erasure function can be found in Appendix C.

4.3 Constructing the domain

We begin by defining our domain of values D. Our domain is made up of two broad classes of values: D_V corresponds to the values that will correspond to terms, and D_T corresponds to type codes, i.e. values that represent types. Following Abel [Abe13] our domain is "untyped" in that we simply construct values that correspond to an untyped lambda calculus (which closely models run-time behaviour of functional programming languages). We will bring typing into our semantics later, by way of the PER model.

A value $d : D_V$ is either: a natural number $n : \mathbb{N}$, a function $a \mapsto b : D \to D$[1], a pair $(a, b) : D \times D$ or an empty list symbol []. Accordingly, a type code $d_t : D_T$ is either: the Nat type code, a dependent function type code, a dependent pair type code, a list type code, a vector type code, or a universe type code (with hidden universe level).

[1]This function notation helps us distinguish semantic functions from syntactic lambdas

$$
\begin{array}{lll}
D & = D_V & \cup \quad D_T \\
D_V & = \mathbb{N} & \cup \quad D \to D \quad \cup \quad D \times D \quad \cup \quad \{[]\} \\
D_T & = \mathtt{Nat}_D & | \quad \mathtt{Vec}_D(D_T, D) \quad | \quad \mathtt{List}_D\, D_T \\
 & \quad | \quad \mathtt{Fun}_D(D_T, D \to D_T) & | \quad \mathtt{Pair}_D(D_T, D \to D_T) \mid \quad \mathtt{Set}_D
\end{array}
$$

Figure 4.2: Values in D

4.4 Evaluating terms into domain values

We now proceed to write a function to evaluate syntactic terms into semantic values. Our function maximally evaluates terms to simplify comparisons. When we enter under a binder, e.g. $\lambda(a : A).b$, we get a function from some value a_v to the interpretation of the body b, where any reference to a in b is replaced by a_v. We can generalize this to any open terms using environments: We define environments γ : $\mathtt{Env}$ as an association of variables to domain values. $\gamma(a)$ is the result of looking up variable a in the environment. $\gamma[a \mapsto a_v]$ extends the environment γ with the variable a mapped to value a_v. Thus, the semantics of an open term is a function from a valid environment to some value in D. We will define what valid environments are later on, for now it suffices to intuitively think of valid environments as binding each variable in the context to an adequate value.

Definition 1. *Terms are evaluated to functions on their environment*

$\gamma : \mathtt{Env}$

$(\!|\_|\!)\_ : P \to \mathtt{Env} \to D$

(*Variables*)	
$(\!\|x\|\!)_\gamma$	$= \gamma(x)$
(*Constructors:*)	
$(\!\|\lambda(a : A).b\|\!)_\gamma$	$= a_v \mapsto (\!\|b\|\!)_{\gamma[a \mapsto a_v]}$
$(\!\|(a, b)\|\!)_\gamma$	$= ((\!\|a\|\!)_\gamma, (\!\|b\|\!)_{\gamma[x \mapsto (\!\|a\|\!)_\gamma]})$
$(\!\|z\|\!)_\gamma$	$= 0$
$(\!\|\mathtt{suc}\ n\|\!)_\gamma$	$= 1 + (\!\|n\|\!)_\gamma$
$(\!\|\mathtt{[]}_l\|\!)_\gamma$	$= []$
$(\!\|\mathtt{cons}_l\, a\, b\|\!)_\gamma$	$= ((\!\|h\|\!)_\gamma, (\!\|t\|\!)_\gamma)$
(*Eliminators*)	
$(\!\|f \cdot a\|\!)_\gamma$	$= (\!\|f\|\!)_\gamma((\!\|a\|\!)_\gamma)$
$(\!\|\mathtt{el} \times a\, P\, b\|\!)_\gamma$	$= (\!\|b\|\!)_{\gamma[x \mapsto x_v, y \mapsto y_v]}$
	where: $(\!\|a\|\!)_\gamma = (x_v, y_v)$
$(\!\|\mathtt{elNat}\, a\, \_\, b\, c\|\!)_\gamma$	$= \mathtt{rec}_{\mathbb{N}}((\!\|a\|\!)_\gamma, (\!\|b\|\!)_\gamma, (k, r)(\!\|c\|\!)_{\gamma[m \mapsto k, p \mapsto r]})$
$(\!\|\mathtt{elList}\, xs\, \_\, b\, c\|\!)_\gamma$	$= \mathtt{rec}_L((\!\|xs\|\!)_\gamma, (\!\|b\|\!)_\gamma, (h, t, r) \mapsto (\!\|c\|\!)_{\gamma[a \mapsto h, as \mapsto t, p \mapsto r]})$

Lambdas are evaluated to functions and accordingly lambda application evaluates to function application in the domain. Natural numbers are mapped to the set[2] of natural numbers $\mathbb{N}$ with the successor operation mapped to a simple addition operation. Recursion on natural numbers is defined by way of a semantic recursion principle $rec_{\mathbb{N}}(a, b, i)$: a is the number being case matched on, b is the value for the base case, i is the inductive step. $i(m, p)$ takes two arguments: the predecessor m and the result of induction p. We obtain this i function by evaluating the syntactic inductive branch c and bind the two syntactic variables to the values given. Elimination on pairs implies splitting the pair into its left and right value and feeding them to the evaluation of the branch argument b.

Definition 2. *Structural induction on* $\mathbb{N}$

$\mathtt{rec}_{\mathbb{N}}(0, b, i)$	$= b$
$\mathtt{rec}_{\mathbb{N}}(1 + m, b, i)$	$= i(m, \mathtt{rec}_{\mathbb{N}}(m, b, i))$

[2]Or type, we are relatively agnostic on the specific formalization

Definition 3. *Structural induction on Lists*

$$\begin{aligned}\mathtt{rec}_L([], b, i) &= b\\ \mathtt{rec}_L((h,t), b, i) &= i(h, t, \mathtt{rec}_L(t, b, i))\end{aligned}$$

Our syntax includes types so we need to give its definition on syntactic type terms. However this is where things can get a bit unclear. We are going to call the result of evaluation on syntactic types: type *codes*. We will not call them type values. Later when we give our PER model we will define type *values*, or semantic types, in terms of PER. Evaluating syntactic types gives us an *encoding* of types in our semantic domain, *not* a semantic type. Because of this we have custom encodings for types, the only interesting one being the encoding for functions.

Definition 4. *Evaluation of types as terms*

$$\begin{aligned}(\!|(a : A) \to B|\!)_\gamma &= \mathtt{Fun}_D((\!|A|\!)_\gamma, a_v \mapsto (\!|B|\!)_{\gamma[a \mapsto a_v]})\\ (\!|(a : A) \times B|\!)_\gamma &= \mathtt{Pair}_D((\!|A|\!)_\gamma, a_v \mapsto (\!|B|\!)_{\gamma[a \mapsto a_v]})\\ (\!|\mathtt{List}\ A|\!)_\gamma &= \mathtt{List}_D\,(\!|A|\!)_\gamma\\ (\!|\mathtt{Nat}|\!)_\gamma &= \mathtt{Nat}_D\\ (\!|\mathtt{Set}|\!)_\gamma &= \mathtt{Set}_D\end{aligned}$$

Our functions are dependent: The codomain B will depend on the domain A. A function type code $\mathtt{Fun}_D(A, F)$ is the type code for its argument type $A \in D$ together with a function from its argument to its result type code $F : D \to D$.

Definition 5 (Interpretation of terms)**.** *Terms are interpreted into their evaluation post-erasure*

$$[\![a]\!]_\gamma = (\!|\downarrow a|\!)_\gamma$$

4.4.1 Including functions in the domain in a well-founded manner

Defining the value of functions is problematic, as the naive presentation we have given $D \to D : D$ is not well founded and leads to circularity. The question is then how to construct the domain of functions in a well founded manner.

Abel [Abe13, section 3.2] illustrates two options (and uses the latter): use domain theory — where values are Scott-continuous functions — or defunctionalize and use closures. We will explain how the two solutions differ and why our results can be reformulated in either approach.

In domain theory we solve the domain equation $D \cong [D \to D]$, where $[D \to D]$ denotes Scott-continuous functions. The approach involves, roughly, constructing successive domains D_0, D_1, D_2, D_3 and defining D as the limit of these successive constructions. This is established work, however defining domain theoretic constructs is quite involved [Abe13, section 3.2]. If we took this approach, we would evaluate lambda terms into semantic functions as such: $(\!|\lambda(a : A).b|\!)_\gamma = a_v \mapsto (\!|b|\!)_{\gamma[a \mapsto a_v]}$.

If we instead take Abel's defunctionalized approach [Abe13, section 3.2], we define our domain to have a closure value $(\gamma, \lambda(a : A).b)$ which pairs a syntactic lambda term with the environment it was evaluated in. We add onto this domain a primitive application operation $App : D \times D \to D$ which computes a closure together with an argument to a domain value. This application operation would be defined in terms of evaluation $App((\gamma, \lambda(a : A).b), a_v) = (\!|b|\!)_{\gamma[a \mapsto a_v]}$.

The only difference in practical terms between either formalization is if we evaluate directly into a semantic function or if we delay evaluation until the argument value is provided. With regards to the relating of function values the rule is the same — related environments and related inputs map to related outputs. For ease of readability we will write function values like functions. Formally we will keep in mind that actually our functions are secretly closures, but would like to remind the reader that it should be possible to reformulate our claims in domain theoretic terms, should they so desire.

4.5 Relating values via semantic types as PER

4.5.1 Semantic types as PER

Semantic types $\mathscr{A} \subseteq D \times D$ are PER on the domain values D, they tell us how to relate values which are of the same "type". We build these relations inductively, implicitly assuming a universe hierarchy we mostly omit for conciseness. We refer to the set of PER and relations on D as $\texttt{Per} \subseteq \texttt{Rel}$.

A PER $\mathscr{A}$ is a relation on some set D that respects transitivity and symmetry, but not reflexivity. This means that for arbitrary $a \in D$ it will not always hold that $a \approx a : \mathscr{A}$. However combining transitivity and symmetry gives us reflexivity; once a value is related to any other value, it is also related to itself: $a \approx b : \mathscr{A} \implies b \approx a : \mathscr{A} \implies a \approx a : \mathscr{A}$. Due to this we can also define a PER $\mathscr{A}$ as a (total) equivalence relation on some subset $A \subset D$. We use $a \in \mathscr{A}$ to mean that a is a member of this subset. More concretely $a \in \mathscr{A}$ if $\exists a'.a \approx a' : \mathscr{A}$.

Intuitively, this means that a semantic type $\mathscr{A}$ does two things simultaneously: specify a subset of well-behaved values and tell us which of these are equivalent. The collection of all of our semantic types is the collection of quotient sets which span all "reasonable" values.

We will now proceed to give a definition of how to construct each of our semantic types, by giving an overview of each type — with an eye towards the fact that each PER respects transitivity and symmetry, arguing the case where it is not clear. We will define each PER in an inductive manner from the ground up, courtesy of our predicative universes [Abe13, section 4.3]. Given the relational style of this semantics we will use the notational shorthand $\forall a \approx a' : \mathscr{A}$ to mean $\forall a, a'$ such that $a \approx a' : \mathscr{A}$.

4.5.2 Semantic types

Let us begin with the semantic type for natural numbers $\mathscr{Nat}$. Assuming we have some denotation for natural numbers $\mathbb{N}$ (set theoretic, type theoretic, etc.), we relate them inductively: 0 relates to itself and successive numbers are related if their predecessors are related. We can prove symmetry and transitivity trivially by induction.

Definition 6 (Semantic type of natural numbers)**.**

$$\frac{}{0 \approx 0 : \mathscr{Nat}} \qquad \frac{n \approx m : \mathscr{Nat}}{1 + n \approx 1 + m : \mathscr{Nat}}$$

Vector and list values are related in a congruence manner. We write (h, n, t) as a shorthand for $(h, (n, t))$ for vector values.

Definition 7 (Semantic type of lists)**.**

$$\frac{}{[] \approx [] : \mathscr{List}(\mathscr{A})} \qquad \frac{h \approx h' : \mathscr{A} \quad t \approx t' : \mathscr{List}(\mathscr{A})}{(h, t) \approx (h', t') : \mathscr{List}(\mathscr{A})}$$

Definition 8 (Semantic type of vectors)**.**

$$\frac{}{[] \approx [] : \mathscr{Vec}(\mathscr{A}, n)} \qquad \frac{h \approx h' : \mathscr{A} \quad n \approx n' : \mathscr{Nat} \quad t \approx t' : \mathscr{Vec}(\mathscr{A}, n)}{(h, n, t) \approx (h', n', t') : \mathscr{Vec}(\mathscr{A}, 1 + n)}$$

We now proceed to the semantic type for dependent functions $\Pi(\mathscr{A}, \mathscr{F})$. $\mathscr{A}$ is the semantic type of the argument and $\mathscr{F} : \mathscr{A} \to \texttt{Per}$ the type family: the function which computes the result type from the argument value. More precisely, $\mathscr{F}$ is a function *on* the relation $\mathscr{A}$, which respects $\mathscr{A}$. It maps related values to the same relation: $\forall a \approx a' : \mathscr{A}.\mathscr{F}(a) = \mathscr{F}(a')$. The extensionality principle induces a PER.

Definition 9 (Semantic function types: $\Pi(\mathscr{A}, \mathscr{F})$)**.** *Given two functions $f, f' \in D \to D$ we state they are related by extensionality; If their outputs are related then they are related.*

$$\frac{\forall a \approx a' : \mathscr{A}. f(a) \approx f(a') : \mathscr{F}(a)}{f \approx f' : \Pi(\mathscr{A}, \mathscr{F})}$$

Dependent pairs have an analogous definition. The difference being that we have input-output pairs and thus do not need to quantify over all possible inputs.

Definition 10 (Semantic dependent product types: $\Sigma(\mathscr{A}, \mathscr{F})$)**.** *Pairs are related pointwise, with dependency in the type specification.*

$$\frac{a \approx a' : \mathscr{A} \quad b \approx b' : \mathscr{F}(a)}{(a, b) \approx (a', b') : \Sigma(\mathscr{A}, \mathscr{F})}$$

We now define semantic universes by way of inductive inference rules and a recursive lifting function $[\_] : D \to \mathscr{Set}$. The inductive rules relate type codes as members of the PER $\mathscr{Set}$. The lifting function "lifts" type codes in D to members of the semantic type $\mathscr{Set}$.

Definition 11 (Semantic type of type codes)**.**

$$\frac{}{\mathtt{Nat}_D \approx \mathtt{Nat}_D : \mathscr{Set}}\ S - \mathtt{Nat}_D \qquad [\mathtt{Nat}_D] = \mathscr{Nat}$$

$$\frac{A \approx A' : \mathscr{Set} \quad n \approx n' : \mathscr{Nat}}{\mathtt{Vec}_D(A, n) \approx \mathtt{Vec}_D(A', n') : \mathscr{Set}}\ S - \mathtt{Vec}_D(,) \qquad [\mathtt{Vec}_D(A, n)] = \mathscr{Vec}([A], n)$$

$$\frac{\begin{array}{c} A \approx A' : \mathscr{Set} \\ a \approx a' : [A] \implies B(a) \approx B'(a') : \mathscr{Set} \end{array}}{\mathtt{Fun}_D(A, B) \approx \mathtt{Fun}_D(A', B') : \mathscr{Set}}\ S - \mathtt{Fun}_D(,) \qquad [\mathtt{Fun}_D(A, B)] = \Pi([A], a \mapsto [B(a)])$$

$$\frac{\begin{array}{c} A \approx A' : \mathscr{Set} \\ a \approx a' : [A] \implies B(a) \approx B'(a') : \mathscr{Set} \end{array}}{\mathtt{Pair}_D(A, B) \approx \mathtt{Pair}_D(A', B') :}\ S - \mathtt{Pair}_D(,) \qquad [\mathtt{Pair}_D(A, B)] = \Sigma([A], a \mapsto [B(a)])$$

$$\frac{i < k}{\mathtt{Set}_i \approx \mathtt{Set}_i : \mathscr{Set}_k}\ S - \mathtt{Set}_D \qquad [\mathtt{Set}_i] = \mathscr{Set}_i$$

We specify the rules for universes only once to show we are still working in a predicative manner and building our semantic types inductively. We will continue to ignore universe levels unless relevant.

Lemma 1 (Related type codes lift to equal PER)**.** *If two type codes are semantically related*

$$A \approx B : \mathscr{Set}$$

Then their lift induce equal PER

$$[A] = [B]$$

Proof. Holds by induction on the rules: Holds trivially for base cases $S - \mathtt{Nat}_D$ and $\mathtt{Set}_D$ and by induction for inductive cases $S - \mathtt{Fun}_D(,)$, $S - \mathtt{Pair}_D(,)$ □

Now we have an evaluation function $(\!|A|\!)_\gamma$ to compute type codes, a specification for relating type codes and a function to lift type codes into semantic types. Putting everything together we define an operation to go from syntactic types to semantic types: Evaluating types to a type code and then lifting these to a semantic type gives us the semantic type associated with that code.

Definition 12 (Interpretation of types)**.** *Types are interpreted to the PER obtained from lifting the type code obtained from evaluation post-erasure*

$$[\![A]\!]_\gamma = [(\!|\downarrow A|\!)_\gamma]$$

4.5.3 Relating environments

We can relate terms and types now in our semantic domain. In order to relate open terms we now also need some mechanism to relate contexts in the semantic realm; we must relate environments $\gamma \approx \gamma' : [\![\Gamma]\!]_\gamma$, in such a way that the provided values respect types bound in the context Γ.

Definition 13 (Interpretation of contexts)**.** *Contexts are interpreted to the PER of related environments*

$$\frac{}{\varnothing \approx \varnothing : [\![\varnothing]\!]}\ S-\varnothing \qquad \frac{\gamma \approx \gamma' : [\![\Gamma]\!] \quad a \approx a' : [\![A]\!]_\gamma}{\gamma[x \mapsto a] \approx \gamma'[x \mapsto a'] : [\![\Gamma, x \overset{\omega}{:} A]\!]}\ S-Ext\omega \qquad \frac{\gamma \approx \gamma' : \Gamma}{\gamma \approx \gamma' : \Gamma, x \overset{0}{:} A}\ S-Ext0$$

4.6 Semantic run-time equivalence

We now have the necessary building blocks to define a semantic run-time equivalence judgement. We can interpret contexts to related environments, terms to domain values and types to PER.

Definition 14. *Terms are semantically run-time equivalent if for any related environments they erase and evaluate to related values.*

$$\Gamma \vDash a = b : A \iff \forall \gamma \approx \gamma' : [\![\Gamma]\!].[\![a]\!]_\gamma \approx [\![b]\!]_{\gamma'} : [\![A]\!]_\gamma \tag{4.1}$$

Chapter 5

Proof

Given that we now have a semantics of erasure and equivalence at run-time we specify our central theorem. We make use of a strong run-time equivalence relation that folds the conditions of correctness of the typing of runid terms into the relation. We explain how to define this relation. From there we give an important lemma that terms related by our strong relation will not fail on erasure, allowing us to exclude such cases from our main proof. Then we introduce the central theorem: (strong) syntactic run-time equivalence implies semantic run-time equivalence.

5.1 Strong run-time equivalence relation

To help our proof we use a modified version of our syntactic relation. Intuitively, this relation unites the typing judgement and previous (weak) relation into a strong relation that closely models the logical relations-style semantics that we use in our proof. This means the relation contains all the information we need to prove correctness without relying on an invariant or reference to typing rules. However it makes checking the relation inefficient as one has to do duplicate checking of already checked runid terms. The strong relation also adds a type annotation which is useful in our proof to determine which semantic type should be used to compare the terms.

We will give an overview of the changes in our strong relation vis-a-vis the weak relation. We give an exhaustive list of the typing rules in Appendix D.

Building the strong relation

Because PER do not have reflexivity in general, we discard the reflexivity rule $\Gamma \vdash a \ \sim_r \ a$. Instead, we add "reflexive" rules for each base term and rely on congruence rules for each constructor. As an example, we give the rules for constructing equivalent `Nat` terms.

$$\frac{}{\Gamma \vdash \mathtt{z} \sim_s \mathtt{z} : \mathtt{Nat}} \qquad \frac{\Gamma \vdash n \sim_s m : \mathtt{Nat}}{\Gamma \vdash \mathtt{suc}\ n \sim_s \mathtt{suc}\ m : \mathtt{Nat}}$$

Figure 5.1: Strong run-time equivalence for `Nat` constructors

We would like to only capture well-formed terms in the relation. In the weak relation this was maintained via typing rules on the side. Now we add an extra precondition to all the rules saying $\Gamma \vdash a \sim_s a : A$ for all subterms (unless the a is already related to some other term in the conditions). Since the only way reflexivity can be formed is via structural rules described above, this captures well-formedness instead of the side conditions.

For equivalence rules that rely on well-typedness of annotations — i.e. erasure and runid — we equate the term with its optimization. Erased functions are equivalent to erased lambdas, and runid functions to identity functions.

$$\frac{\Gamma \vdash f \sim_s \lambda(x \overset{0}{:} A).b : (x \overset{0}{:} A) \to B}{\Gamma \vdash f \overset{0}{\cdot} a \sim_s b : B} \qquad \frac{\Gamma \vdash f \sim_s \lambda_r(x : A).x : (a \overset{\omega}{:} A) \to A \quad \Gamma \vdash a \sim_s a : A}{\Gamma \vdash f \cdot_r a \sim_s a : A}$$

Figure 5.2: Strong run-time equivalence of applications

Another element of this well-formedness is that terms marked as runid are subject to conditions in the weak run-time equivalence. The canonical example being runid lambdas, which have the condition that their bodies are equivalent to their argument. These conditions are maintained in the strong equivalence.

$$\frac{\Gamma \vdash \lambda(x \overset{\omega}{:} A).b \overset{\sigma}{:} (x \overset{\omega}{:} A) \to B \quad \Gamma, x \overset{\omega}{:} A \vdash b \ \sim_r \ x}{\Gamma \vdash \lambda_r(x : A).b \overset{\sigma}{:} (x : A) \to_r B} \vdash \lambda_r$$

$$\frac{\Gamma, a \overset{\omega}{:} A \vdash b \sim_s a : A}{\Gamma \vdash \lambda_r(a : A).b \sim_s \lambda(a : A).a : (a : A) \to_r A}$$

Figure 5.3: Runid Lambdas: typing rule vs strong run-time equivalence rule

5.2 Related terms have defined erasure

Theorem 1 (Related terms have defined erasure)**.** *If $\Gamma \vdash a \sim_s b : A$ then neither a nor b will be a term in erased position, i.e. $\downarrow a \neq \bot$ and $\downarrow b \neq \bot$*

The erasure function fails when the input is a term in erased position, however it does not flow from a run-time term into an erased term. Our syntactic rules follow a similar structure. If we operate by induction we see that none of the axioms will relate a term in erased position, and our inductive rules will not relate the erased portions either.

Proof. We operate by induction on the strong rules:

1. Base cases:
   a) variables x are only ever erased to $\bot$ if $x \overset{0}{:} A \in \Gamma$ which is excluded by the assumption in the variable rule
   b) All other base rules contain no erased fragments and thus erase directly to themselves, i.e. not $\bot$
2. Inductive case: We have the following inductive rules:
   a) Congruence rules, e.g. $\Gamma \vdash \mathtt{inl}^{\omega,\omega} a \sim_s \mathtt{inr}^{\omega,\omega} b : A \uplus B$. These do not erase to $\bot$ if no subterm erases to $\bot$, which holds by induction.
   b) Erasure rules, e.g. $\Gamma \vdash \mathtt{inl}^{\omega,0} a \sim_s b : A \overset{\omega}{\uplus}{}^{0} B$. The erasure function does not subrecurse on erased subterms, and the subterms marked not erased will not reduce to $\bot$ by induction.
   c) Runid rules. This also holds by induction, by analogous logic.

d) Erased constructors, i.e. erased left injections $\mathtt{inl}^{0,\omega}a$ (and erased right injection). Neither of these are included in our relation.

None of the inductive rules relate terms that would erase to $\bot$ so $\downarrow a \neq \bot$.

As such for any $\Gamma \vdash a \sim_s b : A$ in our relation, erasure will succeed for both a and b. □

5.3 Main theorem

Our main theorem is proving the fundamental theorem of logical relations: syntactic run-time equivalence entails semantic run-time equivalence.

Theorem 2. *Syntactic run-time equivalence implies semantic run-time equivalence*
$\Gamma \vdash a \sim_s b : A \implies \Gamma \vDash a = b : A$
By Definition 14 more explicitly:
If a, b are syntactically related $\Gamma \vdash a \sim_s b : A$ then: Interpreting a, b with related environments $\gamma \approx \gamma' : [\![\Gamma]\!]$ produces related terms $[\![a]\!]_\gamma \approx [\![b]\!]_{\gamma'} : [\![A]\!]_\gamma$ in semantic type $[\![A]\!]_\gamma$

Our proof will operate by induction on the strong rules. We will give an overview of the classes of rules with illustrative cases to detail the proof strategy. Each case will show the relevant syntactic rule and operate by induction on the assumptions of the rule. More involved proofs will rely on helper lemmas.

We will implicitly assume arbitrary enviroments $\gamma \approx \gamma' : [\![\Gamma]\!]$, such that every statement quantifying over environments of Γ will use the same environment. We will also implicitly assume the same for context extensions in assumptions. The first few lemmas will manually fix the contexts and extensions and subsequent lemmas will use analogous logic.

We use Theorem 1 to know that we do not have to handle failure of erasure, and can assume that the erasure function succeeds.

We first prove the PER rules subsection 5.3.1, then go over the proof strategy for regular rules subsection 5.3.2, rules for erasure subsection 5.3.3 and finally rules for runid terms subsection 5.3.4. These classes of rules exhaustively cover the set of strong run-time equivalence rules thus the main theorem holds by induction on the strong run-time equivalence rules.

5.3.1 PER rules

Our strong relation still has inference rules for symmetry and transitivity. These can be fairly trivially proven with by combining induction with the PER rules of the semantic domain.

Case 1 (Strong relation is a PER)**.** *Symmetry and transitivity in the strong relation entail the same results in the domain.*

1. *Given $\Gamma \vdash b \sim_s a : A$ it holds that $\Gamma \vdash a \sim_s b : A$*
2. *Given $\Gamma \vdash a \sim_s b : A$ and $\Gamma \vdash b \sim_s c : A$ it holds that $\Gamma \vdash a \sim_s c : A$*

By proving transitivity and symmetry in general we can operate in our proof in an equational manner, rewriting terms.

5.3.2 Unerased rules

Rules relating unmarked and unerased terms are either base reflexivity rules or congruence rules. The proof strategy for these cases is relatively straightforward, axioms in the source map to axioms in the PERs, the conditions of composite semantic types are obtained via induction on the assumptions (which by congruence in the source relation we have for each subterm).

To this end we detail the proof strategy for:

1. Rules on types with function types
2. Rules on constructors with $\texttt{suc}\ \ a$ and $\lambda(a \overset{\omega}{:} A).b$
3. Rules on eliminators with application and induction on nats

Rules on types

The base types $\texttt{Nat}, \texttt{Set}$ are axiomatically related in both the source and semantic domain. Hence these are base cases in our main proof and are trivial.

Dependent functions $(a \overset{\omega}{:} A) \to B$, products $(a \overset{\omega}{:} A) \times B^{\omega}$, lists $\texttt{List}\ A$ and vectors $\texttt{Vec}\ A\ n^{\omega}$ are the inductive cases. We show the proof for functions as the others are analogous.

Case 2 (Unerased function type)**.** *Regular function types are semantically equal if their argument types are equal and their return types are extensionally equal*

$$\frac{\Gamma \vdash A \sim_s C : \texttt{Set} \quad \Gamma, a \overset{\omega}{:} A \vdash B \sim_s D : \texttt{Set}}{\Gamma \vdash (a \overset{\omega}{:} A) \to B \sim_s (c \overset{\omega}{:} C) \to D : \texttt{Set}}$$

Proof. For arbitrary $\gamma \approx \gamma' : \Gamma$ we wish to prove

$$[\![(a \overset{\omega}{:} A) \to B]\!]_{\gamma} \approx [\![(a \overset{\omega}{:} C) \to D]\!]_{\gamma'} : \mathcal{Set}$$

If we evaluate both sides of the relation we are left to prove

$$\texttt{Fun}_D([\![A]\!]_{\gamma}, a_v \mapsto [\![B]\!]_{\gamma[a \mapsto a_v]}) \approx \texttt{Fun}_D([\![C]\!]_{\gamma}, c_v \mapsto [\![D]\!]_{\gamma'[a \mapsto c_v]}) :$$

By Definition 11 for function type codes we require two conditions: argument types relate and return type functions relate by extensionality.

1. $[\![A]\!]_{\gamma} \approx [\![C]\!]_{\gamma'} : \mathcal{Set}$ holds by induction on the first assumption
2. For any $a_v \approx c_v : [\![A]\!]_{\gamma}$ we need to prove that supplying each argument to the type code function gives us related typecodes, i.e.

   $$[\![B]\!]_{\gamma[a \mapsto a_v]} \approx [\![D]\!]_{\gamma'[a \mapsto c_v]} : \mathcal{Set}$$

   Which similarly holds by induction on the second assumption

As both conditions are satisfied our typecodes are equivalent. □

Constructors

Similarly we have base cases and inductive cases.

The base cases in the strong relation are also axioms in the domain, e.g. $\Gamma \vdash \texttt{z} \sim_s \texttt{z} : \texttt{Nat}$ holds since $0 \approx 0 : \mathcal{Nat}$.

Inductive cases are congruence rules and hold similarly by induction on the assumptions of the rules. We give two proofs, one for $\texttt{suc}\ \ a$ and one for $\lambda(a \overset{\omega}{:} A).b$. The former relies on congruence conditions in the semantic domain and thus shows an analogous proof for pairs. The latter shows the technique for functions, which is instructive as all our domain values are either base values or functions on base values. This will become useful when discussing the proofs for eliminators.

Case 3 (Congruence on suc)**.**

$$\frac{\Gamma \vdash n \sim_s m : \mathit{Nat}}{\Gamma \vdash \mathit{suc}\ \ n \sim_s \mathit{suc}\ \ m : \mathit{Nat}}$$

Proof. We need to prove that, for some $\gamma \approx \gamma' : [\![\Gamma]\!]$,

$$[\![\mathsf{suc}\ a]\!]_\gamma \approx [\![\mathsf{suc}\ b]\!]_{\gamma'} : \mathcal{N}at$$

By evaluation we need to prove

$$1 + [\![a]\!]_\gamma \approx 1 + [\![b]\!]_{\gamma'} : \mathcal{N}at$$

By congruence on semantic Nats (Definition 6) it suffices to prove

$$[\![a]\!]_\gamma \approx [\![b]\!]_{\gamma'} : \mathcal{N}at$$

Which holds by induction on the assumption. □

Case 4 (Regular lambdas)**.** *Lambdas are related if their bodies are related.*

$$\frac{\Gamma, a \overset{\omega}{:} A \vdash b \sim_s b' : B}{\Gamma \vdash \lambda(a \overset{\omega}{:} A).b \sim_s \lambda(a \overset{\omega}{:} A).b' : (a \overset{\omega}{:} A) \to B}$$

Proof. For arbitrary $\gamma \approx \gamma' : [\![\Gamma]\!]$ we need to prove that

$$(a_v \mapsto [\![b]\!]_{\gamma[a \mapsto a_v]}) \approx (a'_v \mapsto [\![b']\!]_{\gamma'[a \mapsto a'_v]}) : \Pi([\![A]\!]_\gamma, a_v \mapsto [\![B]\!]_{\gamma'[a \mapsto a_v]})$$

By function extensionality (Definition 9) this entails proving that function outputs are related for arbitrary related inputs $a_v \approx a'_v : [\![A]\!]_\gamma$, i.e.

$$[\![b]\!]_{\gamma[a \mapsto a_v]} \approx [\![b']\!]_{\gamma'[a \mapsto a'_v]} : [\![B]\!]_{\gamma[a \mapsto a_v]}$$

Which holds by induction on the assumption. □

Eliminators

Eliminator cases are all inductive rules, we give two cases: function application and elimination on nats. Function elimination shows how to apply the extensionality principle, non-inductive eliminators evaluate to regular function applications so are proven analogously to the case for application. Inductive eliminators differ in their proof as they operate by induction in the semantic domain, as such we give the Nat eliminator case as a simple example of how to prove such cases.

Case 5 (Regular function application)**.**

$$\frac{\Gamma \vdash f \sim_s f' : (x \overset{\omega}{:} A) \to B \quad \Gamma \vdash a \sim_s a' : a}{\Gamma \vdash f \overset{\omega}{\cdot} a \sim_s f' \overset{\omega}{\cdot} a' : B}$$

Proof. By Definition 9, related functions map related terms to related outputs

$$[\![f]\!]_\gamma([\![a]\!]_\gamma) \approx [\![f']\!]_\gamma([\![a']\!]_\gamma) : [\![B]\!]_\gamma$$

Induction on the assumptions tell us that the functions and inputs are related. □

In order to prove the nat eliminator case we need two lemmas: one which lets us know that the interpetation of the motive into a semantic type family produces valid types, and another giving the conditions that need to be satisfied to inductively prove that two recursors on nat values are equivalent.

To know that a semantic type family is valid we rely on the same conditions placed on the type family $\mathscr{F}$ in the semantic function type $\Pi(\mathscr{A}, \mathscr{F})$: i.e. that it respects $\mathscr{A}$.

Lemma 2 (Motives induce valid semantic type family)**.** *Valid motives are valid semantic type families.*
Given

$$P \approx P' : \Pi(\mathscr{A}, \mathscr{S}\mathscr{e}\mathscr{t})$$

The semantic family $\mathscr{P} = a \mapsto [P(a)]$ *is valid, i.e.* $\forall a \approx a' : \mathscr{A}$

$$\mathscr{P}(a) = \mathscr{P}(a')$$

Proof. By the assumption and Definition 9 we know the function P maps related inputs to related outputs. Since the outputs are type codes Lemma 1 tells us these type codes lift to equal semantic types. As such $\mathscr{P}$ is a valid semantic type family. □

Inductive eliminators

Since inductive eliminators are defined semantically via purpose built recursive functions it is useful to first abstract over the specifics and show the conditions necessary to prove two recursive functions on nats equivalent.

Lemma 3 (Equivalence of recursors on natural numbers)**.** *Recursors on natural numbers are equivalent if they are piecewise equivalent in the arguments.*
Given

$$a \approx a' : \mathscr{N}\mathscr{a}\mathscr{t}$$
$$b_v \approx b'_v : \mathscr{P}(z)$$
$$c \approx c' : \Pi(\mathscr{N}\mathscr{a}\mathscr{t}, n \mapsto \Pi(\mathscr{P}(n), \_ \mapsto \mathscr{P}(1+n)))$$

It holds that

$$\mathtt{rec}_{\mathbb{N}}(a, b, c) \approx \mathtt{rec}_{\mathbb{N}}(a', b', c') : \mathscr{P}(a)$$

Proof. We operate by induction on $a \approx a' : \mathscr{N}\mathscr{a}\mathscr{t}$

1. Base case: $0 \approx 0 : \mathscr{N}\mathscr{a}\mathscr{t}$.
   We need to prove that
   $$\mathtt{rec}_{\mathbb{N}}(0, b, c) \approx \mathtt{rec}_{\mathbb{N}}(0, b', c') : \mathscr{P}(0)$$
   Which computes to
   $$b \approx b' : \mathscr{P}(0)$$
   Which holds by assumption.
2. Inductive case: $1 + k \approx 1 + k' : \mathscr{N}\mathscr{a}\mathscr{t}$.
   Given the induction hypothesis:
   $$\mathtt{rec}_{\mathbb{N}}(k, b, c) \approx \mathtt{rec}_{\mathbb{N}}(k', b', c') : \mathscr{P}(k)$$
   We need to prove that
   $$\mathtt{rec}_{\mathbb{N}}(1 + k, b, c) \approx \mathtt{rec}_{\mathbb{N}}(1 + k', b', c') : \mathscr{P}(1 + k)$$
   If we compute our proof term we get
   $$c(k, \mathtt{rec}_{\mathbb{N}}(k, b, c)) \approx c(k', \mathtt{rec}_{\mathbb{N}}(k', b', c')) : \mathscr{P}(1 + k)$$
   By assumption it holds that
   $$c \approx c' : \Pi(\mathscr{N}\mathscr{a}\mathscr{t}, n \mapsto \Pi(\mathscr{P}(n), \_ \mapsto \mathscr{P}(1+n)))$$
   Meaning c, c' are extensionally related: related inputs are mapped to related outputs. Both inputs are related so the functions must map to related outputs thus the inductive case holds.

As base and inductive case hold, it holds for all $a \approx a' : \mathcal{Nat}$. □

We can now proceed to concretely prove the case for equivalent Nat eliminators

Case 6 (Nat eliminator)**.**

$$\frac{\Gamma \vdash a \sim_s a' : \mathtt{Nat} \qquad \Gamma \vdash P \sim_s P' : (n \overset{\omega}{:} \mathtt{Nat}) \to \mathtt{Set} \\ \Gamma \vdash b \sim_s b' : P \cdot z \quad \Gamma, m \overset{\omega}{:} \mathtt{Nat}, p \overset{\omega}{:} P \cdot m \vdash c \sim_s c' : P \cdot (\mathtt{suc}\ m)}{\Gamma \vdash \mathtt{elNat}\, a\, P\, b\, c \sim_s \mathtt{elNat}\, a'\, P'\, b'\, c' : P \cdot a}$$

Proof. We need to prove that the recursors are equivalent, using Lemma 2 and the second assumption to know the semantic types $\mathscr{P}(\_)$ are valid PER

$$\mathtt{rec}_{\mathbb{N}}([\![a]\!]_\gamma, [\![b]\!]_\gamma, (k, r) \mapsto [\![c]\!]_{\gamma[m \mapsto k, p \mapsto r]}) \approx \mathtt{rec}_{\mathbb{N}}([\![a']\!]_{\gamma'}, [\![b']\!]_{\gamma'}, (k', r') \mapsto [\![c']\!]_{\gamma'[m \mapsto k', p \mapsto r']}) : \mathscr{P}([\![a]\!]_\gamma)$$

From here we can use Lemma 3 which has 3 conditions. Each condition aligns with induction on an assumption

1. The first condition holds by induction on the first assumption

$$[\![a]\!]_\gamma \approx [\![a']\!]_{\gamma'} : \mathcal{Nat}$$

2. the second condition holds by induction on the third assumption

$$[\![b]\!]_\gamma \approx [\![b']\!]_{\gamma'} : \mathscr{P}(0)$$

3. The third condition holds by induction on the third assumption, by extensionality (Definition 9). For arbitrary $k \approx k' : \mathcal{Nat}$, $r \approx r' : \mathscr{P}(k)$ the functions are related.

$$[\![c]\!]_{\gamma[m \mapsto k, p \mapsto r]} \approx [\![c']\!]_{\gamma[m \mapsto k', p \mapsto r']} : \mathscr{P}(1 + k)$$

□

5.3.3 Erased terms

When rules contain erased subterms we know that erasure will map the parent term to some term without erased subterms. Because of this the cases on such rules will involve mapping to a statement on the interpretation of the equivalent erased term. After erasure the relevant statement to prove will align with a case in the previous section and be proven in an analogous manner.

To this end we give the cases on functions with erased argument for a type. Then on erased vec cons operation for a constructor. Then on the right erased pair eliminator.

Types

Erased functions show us how the binding of erased terms and weakening of a context by an erased term produces semantically identical statements. This case is handled analogously in dependent product types with erasure.

Case 7 (Erased function type)**.** *Erased function types map to the same value as their return type weakened by the argument.*

$$\frac{\Gamma, a \overset{0}{:} A \vdash B \sim_s D : \mathtt{Set}}{\Gamma \vdash (a \overset{0}{:} A) \to B \sim_s D : \mathtt{Set}}$$

Proof. We wish to prove that

$$[\![(a \overset{0}{:} A) \to B]\!]_\gamma \approx [\![D]\!]_{\gamma'} : \mathcal{Set}$$

Since $\downarrow (a \overset{0}{:} A) \to B = \downarrow B$ it suffices to show that

$$[\![B]\!]_\gamma \approx [\![D]\!]_{\gamma'} : \mathcal{Set}$$

Which we directly obtain by induction on the first condition of the rule.

□

Constructors with erasure

Constructors with erased content operate analogously to the previous congruence rules, but ignoring erased portions. We give the lemma for non-empty length-erased vectors as an example.

Case 8 (Erased vec cons)**.** *Erased nonempty vectors relate to their head and tail in a list*

$$\frac{\Gamma \vdash h \sim_s h' : A \quad \Gamma \vdash t \sim_s t' : \mathsf{Vec}\, A\, n^0}{\Gamma \vdash \mathsf{cons}_v^0\, h\, n\, t \sim_s \mathsf{cons}_l\, h'\, t' : \mathsf{Vec}\, A\, \mathsf{suc}\ \ n^0}$$

Proof. We want to prove, after erasing, that

$$([\![h]\!]_\gamma, [\![t]\!]_\gamma) \approx ([\![h']\!]_\gamma, [\![t']\!]_\gamma) : \mathcal{List}(\mathcal{A}) \tag{5.1}$$

By definition lists are related if they are structurally related so we need to prove the head and tail are related

1. $[\![h]\!]_\gamma \approx [\![h']\!]_{\gamma'} : \mathcal{A}$ Holds by the first assumption
2. $[\![t]\!]_\gamma \approx [\![t']\!]_{\gamma'} : \mathcal{List}(\mathcal{A})$ holds by the second assumption

Thus the original statement holds.

□

Erased eliminator

We give the cases on erased application and show how to deal with data type eliminators with the example of a right-erased pair eliminator.

Case 9 (Erased application)**.**

$$\frac{\Gamma \vdash f \sim_s \lambda(x \overset{0}{:} A).b : (x \overset{0}{:} A) \to B}{\Gamma \vdash f \overset{0}{\cdot} a \sim_s b : B}$$

Proof. We need to prove that erased function applications are equivalent to their bodies

$$[\![f \overset{0}{\cdot} a]\!]_\gamma = [\![f]\!]_\gamma \approx [\![b]\!]_{\gamma'} : [\![B]\!]_\gamma$$

Which holds by induction on the assumption

$$[\![f]\!]_\gamma \approx [\![\lambda(x \overset{0}{:} A).b]\!]_{\gamma'} = [\![b]\!]_{\gamma'} : [\![B]\!]_\gamma$$

□

Erased data eliminators operate by changing the nominal type, for sums and pairs this involves mapping to an applied let binding. We show how to do this for right erased pairs as an illustrative example

Case 10 (Right-erased pair eliminator)**.** *Given*

$$\frac{\Gamma \vdash p \sim_s p : A \quad \Gamma \vdash P \sim_s P : (z \overset{\omega}{:} (x \overset{\omega}{:} A) \times B^0) \to \mathtt{Set} \qquad \Gamma, x \overset{\omega}{:} A, y \overset{0}{:} B \vdash b \sim_s c : P \cdot x}{\Gamma \vdash \mathtt{el}^{\omega \times^0} p\, P\, b \sim_s \lambda(a \overset{\omega}{:} A).c \overset{\omega}{\cdot} p : P \cdot a}$$

Proof. We first use Lemma 2 to show that the semantic types obtained from $\mathscr{P} = a_v \mapsto [\![P]\!]_{\gamma[z \mapsto a_v]}$ are valid.
We use the definition of erasure on right-erased pair eliminators

$$\downarrow \mathtt{el}^{\omega \times^0} p\, P\, b = \lambda(a : A).\downarrow b \cdot \downarrow p$$

to simplify our proof to a function application on both sides

$$[\![\lambda(a : A).b]\!]_\gamma([\![p]\!]_\gamma) \approx [\![\lambda(a : A).c]\!]_{\gamma'}([\![p]\!]_{\gamma'}) : \mathscr{P}(a)$$

The first assumption tells us the inputs are related so by extensionality (Definition 9) it suffices to prove the functions as related:

$$[\![\lambda(a : A).b]\!]_\gamma \approx [\![\lambda(a : A).c]\!]_{\gamma'} : \Pi(\mathscr{A}, x \mapsto \mathscr{P}(x))$$

Which holds by the third assumption with analogous logic to before. □

5.3.4 Runid terms

Most rules on runid terms are analogous to the cases on the unerased variant, as erasure only removes the runid marking. We show the case of runid application as an example of such cases and as the central optimization. The interesting cases are inductive eliminators, so we give the case on inductive runid nat eliminators.

Application

Case 11 (runid application)**.**

$$\frac{\Gamma \vdash f \sim_s \lambda_r(x : A).x : (a \overset{\omega}{:} A) \to A \quad \Gamma \vdash a \sim_s a : A}{\Gamma \vdash f \cdot_r a \sim_s a : A}$$

Proof. We need to prove that applying a runid function is equivalent to its argument

$$[\![f]\!]_\gamma([\![a]\!]_\gamma) \approx [\![a]\!]_{\gamma'} : [\![A]\!]_\gamma$$

Let $id = x \mapsto x$, we can trivially show that $id([\![a]\!]_\gamma) \approx [\![a]\!]_{\gamma'} : [\![A]\!]_\gamma$, so it suffices to prove that

$$[\![f]\!]_\gamma([\![a]\!]_\gamma) \approx id([\![a]\!]_{\gamma'}) : [\![A]\!]_\gamma$$

We use the extensionality principle on functions (Definition 9): related functions take related inputs to related outputs.

1. We show that the functions are related

   $$[\![f]\!]_\gamma \approx id : \Pi([\![A]\!]_\gamma, [\![A]\!]_\gamma)$$

   Since $[\![\lambda_r(x : A).x]\!]_{\gamma'} = x \mapsto x$ this holds by induction on the first assumption.

2. $[\![a]\!]_\gamma \approx [\![a]\!]_{\gamma'} : [\![A]\!]_\gamma$ holds by induction on the second assumption

By showing that the functions and inputs are related, we prove that the applications are related. □

Inductive runid eliminators

Inductive eliminators require a stronger semantic lemma. We need to express the substitution of recursive subterms for recursive subcalls, e.g. $b[p \mapsto m]$ for nats. Because of this we give a stronger semantic lemma with such an induction hypothesis baked in.

Lemma 4 (Runid Nat recursor)**.** *Induction on runid Nats*
Given

$$a \approx a' : \mathcal{Nat}$$

$$b \approx 0 : \mathcal{Nat}$$

$$\mathtt{rec}_{\mathbb{N}}(k, b, i) \approx k' : \mathcal{Nat} \implies i(k, k') \approx 1 + k' : \mathcal{Nat}$$

$$i \approx i : \Pi(\mathcal{Nat}, \Pi(\mathcal{Nat}, \mathcal{Nat}))$$

It holds that

$$\mathtt{rec}_{\mathbb{N}}(a, b, i) \approx a' : \mathcal{Nat}$$

Proof. We proceed by induction on $a \approx a' : \mathcal{Nat}$

1. Base case: $0 \approx 0 : \mathcal{Nat}$
   We need to prove that
   $$\mathtt{rec}_{\mathbb{N}}(0, b, i) \approx 0 : \mathcal{Nat}$$
   If we compute the left term we need to prove
   $$b \approx 0 : \mathcal{Nat}$$
   Which holds by assumption

2. Inductive step: $1 + k \approx 1 + k' : \mathcal{Nat}$
   Our induction hypothesis is
   $$\mathtt{rec}_{\mathbb{N}}(k, b, i) \approx k' : \mathcal{Nat}$$
   We need to prove
   $$\mathtt{rec}_{\mathbb{N}}(1 + k, b, i) \approx 1 + k' : \mathcal{Nat}$$
   If we compute the left term we need to prove
   $$i(k, \mathtt{rec}_{\mathbb{N}}(k, b, i)) \approx 1 + k' : \mathcal{Nat} \tag{5.2}$$
   Using the induction hypothesis and the third assumption we get
   $$i(k, k') \approx 1 + k' : \mathcal{Nat} \tag{5.3}$$
   The fourth assumption lets us use extensionality for i, if the first and second arguments are related the output is related. $k \approx k : \mathcal{Nat}$ holds by assumption, using the induction hypothesis for the second argument we arrive at
   $$i(k, \mathtt{rec}_{\mathbb{N}}(k, b, i)) \approx i(k, k') : \mathcal{Nat} \tag{5.4}$$
   By transitivity, combining Equation 5.3 and Equation 5.4 gives us the exact proof statement we need in Equation 5.2

As the statement holds for the base and inductive case, it holds for all a. □

Since inductive runid eliminators have the aforementioned substitution to express the induction hypothesis in the syntactic rule we need to investigate how sytactic substitutions map to the semantic domain. The following lemma shows us that substitution is equivalent to extending the environment, mapping the variable to the interpretation of the term being substituted for.

Lemma 5. *Evaluation of substitution* $[\![b[a \mapsto c]]\!]_\gamma = [\![b]\!]_{\gamma[a \mapsto [\![c]\!]_\gamma]}$

Proof. We analyze the left and right terms:

- On the left hand side: When we reach a term that is a in b we instead find the term c which is evaluated to $[\![c]\!]_\gamma$
- On the right hand side : When we reach the variable a in b we give the value $\gamma(a) = [\![c]\!]_\gamma$.

The values are the same barring the terms mentioned, and the terms mentioned evaluate to the same value. □

We can now give the case for inductive runid nat eliminators.

Case 12 (Runid Nat eliminator)**.**

$$\frac{\begin{array}{c}\Gamma_1, a \overset{\omega}{:} \mathtt{Nat}, \Gamma_2 \vdash a \sim_s a : \mathtt{Nat} \\ \Gamma_1, \Gamma_2 \vdash b[a \mapsto \mathtt{z}] \sim_s \mathtt{z} : \mathtt{Nat} \\ \Gamma_1, \Gamma_2, m \overset{\omega}{:} \mathtt{Nat} \vdash c[p \mapsto m][a \mapsto \mathtt{suc}\ \ m] \sim_s \mathtt{suc}\ \ m : \mathtt{Nat} \\ \Gamma_1, a \overset{\omega}{:} \mathtt{Nat}, \Gamma_2, m \overset{\omega}{:} \mathtt{Nat}, p \overset{\omega}{:} \mathtt{Nat} \vdash c \sim_s c : \mathtt{Nat}\end{array}}{\Gamma_1, a \overset{\omega}{:} \mathtt{Nat}, \Gamma_2 \vdash \mathtt{el}_r\mathtt{Nat}\, a\, P\, b\, c \sim_s a : \mathtt{Nat}}$$

Proof. Let $a_v = \gamma(a), a'_v = \gamma'(a)$, we need to prove that recursion on the left value is equivalent to the right value

$$\mathtt{rec}_{\mathbb{N}}(a_v, [\![b]\!]_{\gamma[a \mapsto a_v]}, (k, r) \mapsto [\![c]\!]_{\gamma[a \mapsto a_v, m \mapsto k, p \mapsto r]}) \approx a'_v : \mathcal{N}at$$

We use Lemma 4. Which has four conditions, using Lemma 5 to interpret the substitutions:

1. $a_v \approx a'_v : \mathcal{N}at$ holds by the first assumption
2. $[\![b]\!]_{\gamma[a \mapsto 0]} \approx 0 : \mathcal{N}at$ holds by the second assumption
3. For the third condition we say that given

$$\mathtt{rec}_{\mathbb{N}}(k, [\![b]\!]_{\gamma[a \mapsto k]}, (n, r) \mapsto [\![c]\!]_{\gamma[a \mapsto k, m \mapsto n, p \mapsto r]}) \approx k' : \mathcal{N}at \tag{5.5}$$

   We must prove that

$$[\![c]\!]_{\gamma[[a \mapsto k, m \mapsto k, p \mapsto k]} \approx 1 + k' : \mathcal{N}at \tag{5.6}$$

   Which holds by induction on the third assumption

4. The fourth condition tells us the the function is well formed for arbitrary inputs $k \approx k' : \mathcal{N}at, r \approx r' : \mathcal{N}at$

$$[\![c]\!]_{\gamma[a \mapsto k, m \mapsto k, p \mapsto r]} \approx [\![c]\!]_{\gamma'[a \mapsto k', m \mapsto k', p \mapsto r']} : \mathcal{N}at$$

   This follows from the fourth assumption, by analogous logic to Case 6.

As all condition holds the original statement holds by Lemma 4 □

5.4 Limitations

We have given a proof sketch as opposed to a formalized proof. If we had more time we would have presented this proof overview and mechanized the proof for increased rigour.

5.4.1 Polymorphic functions

Our analysis rejects erased parametric functions e.g. $id : (A \overset{0}{:} \texttt{Set}) \to (a \overset{\omega}{:} A) \to A$, because the type variable A no longer exists in our context. However accounting for this would not make our proof or semantics more useful, we could either:

1. Require types with erased parameters to be monomorphized, either by the compiler or metatheoretically (one can regard the semantics or proof as quantifying over the specific type)
2. Build monomorphization into the semantics, interpret the semantic type into an explicit quantification over valid semantic types.
3. Define a different erasure function/semantics for types, and only erase type parameters at the type level after this compiler pass.

Any of these options would gain us more technical complexity with equivalent generality or results. As such our semantics simply exclude such functions and assumes they are monomorphized before being passed to this compiler stage.

5.4.2 Going from the weak to strong relation

Our strong relation morally captures the behaviour of typing and weak relation. Proving this would entail proving soundness of the strong relation with regards to the typing and weak relation. This would take two lemmas:

1. Prove that if $\Gamma \vdash a \overset{\omega}{:} A$, $\Gamma \vdash b \overset{\omega}{:} B$ and $\Gamma \vdash a \sim_r b$ then we can select either A or B for the strong relation. $\Gamma \vdash a \sim_s b : A$ and $\Gamma \vdash a \sim_s b : B$ should still satisfy the semantics.
2. Prove that the weak relation $\Gamma \vdash a \sim_r b$ conditions in the typing rules $\Gamma \vdash a \overset{\omega}{:} A$ entail the strong relation $\Gamma \vdash a \sim_s b : A$

We leave this soundness proof as future work which is out of scope of the current thesis.

Chapter 6

Related work

We will overview prior and related work in the field and how our contributions interact with this work. As we have already specified the primary related work is ornament-based approaches. We will first go over ornaments, custom representations and then agda2hs. We will then briefly go over work in Rocq and compilers.

6.1 Ornaments

Ornaments [McB10] specify a theoretical basis for how to give an algorithm (an algebra) to create one type from another via extension. Using this approach we can, for example, define vectors as an extension of lists: $Vec(A, n) = \{l : \texttt{List}(A) \mid \texttt{length}(A, a) = n\}$, i.e. a list together with a condition on the index (the index equals the length of the list). This encoding of inductive families as dependent pairs is common in the literature. It is used in "Custom Representations" [TB25] and Agda2HS[1] [Coc+22]. We will now elaborate more on these two approaches and how our work interacts with theirs.

In both cases we will emphasize that while the framework gives strong theoretical results, they are both unergonomic in similar ways. Both approaches carry a cost to the programmer from needing refinement types. Agda2HS greatly limits support of runid functions to only support types that are encoded as erased dependent pairs. Theocharis and Brady [TB25] support more types than just refinement types, but require manually coded mappings from existing types to their refinements, including coherence proofs.

6.1.1 Custom representations

Theocharis and Brady [TB25] give a type theory for reasoning about the run-time representation of data generally; decoupling it from the theoretical structure of data. This is motivated by efficiency concerns. For example, natural numbers are formalized in a linked list structure `suc (suc (suc zero)) = 3`. This lets us define operations like addition and theorems with the use of induction, a powerful mathematical tool. Computers however are famously good at representing numbers and doing calculations efficiently with binary representations and operations, not with inefficient linked lists. Dependent languages tend to have built in conversion from Nat primitives to binary representation. Theocharis and Brady [TB25] generalize this to give any data type a run-time representation and trivialize mappings between the different compile time views of the same run-time data.

However as we alluded to earlier this carries some programming cost. We have to give a mapping to a user-defined ornament encoding. This ornament encoding contains user-provided equality proofs.

[1] Which is not explicitly based on ornaments

Let us stick with the example of vectors. We start with traditional inductively defined vector `Vec A n` and list `List A` types. Now we must define the ornamental encoding of vectors `Vec' A n = ∃ (List A) (λ l → length l ≡ n)`. We need to define constructor functions `nil' = (nil, refl)` and `cons' x (xs, p) = (cons x xs, cong succ p)`. Note how to give the constructor we must construct the proof of `length (cons x xs) ≡ succ (length xs)`. Next we need to define the eliminator for `Vec'`, i.e. `elim-Vec' P b r n xs`. We skip the type signature and the definition for the sake of conciseness. The point is that in this definition we must also provide explicit equality proofs. But we are not done yet. We still need to prove our specification of the eliminator correct. We prove its behaviour correct for each constructor. For the nil branch we prove `elimVec' P b r zero (nil', refl) ≡ b`, the statement for the cons branch is more convoluted thus omitted.

Dependent pattern matching can resolve some of these proofs but not all. The proofs are not particularly complex for an experienced prover but still tedious. Using their system as it currently exists implies a large amount of small changes to a large codebase.

6.1.2 Transparent in agda2hs

Agda2HS is less principled in its treatment of runid functions. It has a rudimentary analysis of correctness, and no mechanism to specify representations. Functions are annotated as runid when given the `transparent` pragma. Agda2HS checks correctness by compiling each argument and body and comparing them. It does this at each pattern match. Types are equal at run-time if they literally compile to the same value, not just if they are structurally equal.

This means we cannot trivially equate vectors and lists: The vector constructors and list constructors have nominally different names so are treated as different. Instead, as Listing 6 shows, we must define vectors as a `boxed` type: list plus an index proof, just like the refinement types of ornaments. The programmer is tasked with separating the existing type into data plus proof. Previously `n` would implicitly have the correct value, now we need to provide manual proofs that the index lines up with the data, i.e. the `length xs ≡ n` predicate in the example.

```
Vec : Set → Nat → Set
Vec a n = ∃ (List a) (λ xs → length xs ≡ n)
{-# COMPILE Agda2HS Vec inline #-}

pattern []v = [] ⟨ refl ⟩

_::v_ : a → Vec a n → Vec a (suc n)
x ::v (xs ⟨ p ⟩) = (x :: xs) ⟨ cong suc p ⟩
{-# COMPILE Agda2HS _::v_ inline #-}

listToVec : (xs : List a) → Vec a (lengthNat xs)
listToVec xs = xs ⟨ refl ⟩
{-# COMPILE Agda2HS listToVec transparent #-}
```

Listing 6: listToVec function in Agda2HS

This explicit separation of data into run-time data plus proof makes the analysis of the mapping "trivial", but greatly restricts what a runid function can be or what types can be compiled away. This is specially a problem for languages like Agda that do not tend to encode data types like this. Making use of a feature like this would require a large refactor of existing codebases or large changes to the implementation of Agda. We would instead like a fine-grained approach that integrates well with existing language design and minimizes effort on the part of the programmer.

6.2 Proof irrelevance

6.2.1 Rocq

Some languages, like Rocq, avoid some of these run-time costs, by way of a split type-system. A type-level separation of code and proofs are encoded [Let02]. Proof statements are members of the `Prop` type and code is a member of `Type`. So a compiler could simply disregard members of `Prop`. However, this is an under-approximation with low granularity. There are some things you would want to disregard that you cannot put in `Prop` . We would like a more granular system than what Rocq gives us.

For example Listing 7 shows an attempt to encode a safe head function on vectors where their length index is erased. Everything is fine until we get to defining the `vhead` function, which Rocq will not allow us to do since `nat` lives in `Prop` and thus cannot be eliminated on.

```
Inductive list (A:Set) : Set :=
| nil : list A
| cons : A -> list A -> list A.

Inductive nat : Prop :=
| z : nat
| S :  nat -> nat.

Inductive vec (A : Set) : (nat : Prop) -> Set :=
| nilv : vec A z
| consv : forall (h:A) (n : nat), vec A n -> vec A (S n).

Definition vhead {A : Set} {n : nat} (v : vec A (S n)) : A :=
  match v with
  | cons _ x _ => x
  end.
```

Listing 7: Rocq attempt, vhead function rejected

By comparison in languages with QTT, like Agda, we can define such a function as in Listing 8. This is because Agda uses the index in type checking, and can recognize it is not used in run-time position.

```
data Vec (A : Set) : @0 ℕ → Set where
    [] : Vec A 0
    _::_ : ∀ {@0 n} → A → Vec A n → Vec A (suc n)

vhead : ∀ {n A} → Vec A (suc n) → A
vhead (x :: v) = x
```

Listing 8: Agda vhead function with erasure

6.2.2 Ghost type theory

Ghost type theory [Win24] try to alleviate these problems by way of a universe of "Ghost" types, which behave closer to run-time irrelevance. These can be used for elimination into irrelevant types but also to discard impossible branches, thus enabling us to define the `vhead` function earlier.

Additionally they give a syntactic erasure mapping which extends types with an "impossible" value and inserts an absurd case into cases which are impossible on the basis of ghost values. Future work could try integrating this into our semantics, however it is not clear to what degree they are compatible as they seem to rely on proof-irrelevance and an extensional type theory, while we do not support proof-irrelevance and are intensional.

6.3 Identity function detection in compiler backends

Some compilers already perform identity optimisations for commonplace shapes; Idris 2 recognizes identity functions on List-shaped things [Sta25], among others. However this does not give a structured solution for general types, nor give the programmer any mechanism to assure that this optimization will kick in.

[Boi21] shows an approach to a similar problem in OCaml. It describes identifying identity functions in that it analyses the representation of data types and uncovers that the function does not produce a representationally different value of the input. It is thus in its aims quite close to our system. However some important distinctions is that it isnt a dependently typed language and does not support erasure.

Chapter 7

Conclusion and Future work

This thesis presented a foundational exploration of erasure and run-time identity in a dependently typed setting. We introduced a core type theory enriched with erasure annotations to distinguish between run-time-relevant and run-time-irrelevant components of a program and runid markers to denote redundant computation. To reason about when erased terms remain behaviorally equivalent, we defined a syntactic run-time equivalence relation and gave it a semantics via a partial equivalence relation (PER) model based on NbE [ACD07].

Our erasure semantics were defined as a translation from the source language to a run-time subset, allowing us to precisely characterize which components are retained during execution. We then established the soundness of a strengthened form of the run-time equivalence relation with respect to the erasure and run-time equivalence semantics, proving that erased terms related syntactically are also semantically equal after erasure.

While these results provide a solid foundation, several limitations point toward avenues for future work. Most notably, our system lacks full support for inductive families. Extending our semantics and syntactic framework to handle such families would be an important step toward a general theoretical basis. We have suggested intuitions to define this, primarily on the basis of analysis of signatures via our run-time equivalence behaviour.

Another open challenge lies in the treatment of higher-order runid functions and assumptions generally. Our current approach relies on substitution to encode assumptions about run-time equivalence, but this technique lacks a general and principled justification. A more robust framework for assumptions could resolve this. We attempted to design a more principled solution for this, but ran into problems with positivity and well-foundedness.

Finally, our proof and semantics focus on a core subset of the language and have not been mechanized. Formalizing the development in Agda or a similar proof assistant would not only increase confidence in the results but also help clarify the boundaries of the system and guide further extensions.

Together, these results demonstrate how dependent types can support modalities on run-time behaviour. They form a basis for understanding and optimizing dependently typed programs, and point the way toward richer type systems with verified run-time behavior.

Bibliography

[Abe13] Andreas Abel. "Normalization by evaluation: Dependent types and impredicativity". In: *Habilitation. Ludwig-Maximilians-Universität München* (2013).

[ACD07] Andreas Abel, Thierry Coquand, and Peter Dybjer. "Normalization by Evaluation for Martin-Lof Type Theory with Typed Equality Judgements". In: *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*. IEEE Computer Society, 2007, pp. 3–12. DOI: 10.1109/LICS.2007.33. URL: http://doi.ieeecomputersociety.org/10.1109/LICS.2007.33.

[Atk18] Robert Atkey. "Syntax and Semantics of Quantitative Type Theory". In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*. Ed. by Anuj Dawar and Erich Grädel. ACM, 2018, pp. 56–65. DOI: 10.1145/3209108.3209189. URL: http://doi.acm.org/10.1145/3209108.3209189.

[Boi21] Leo Boitel. *Detecting identity functions in Flambda*. https://ocamlpro.com/blog/2021_07_16_detecting_identity_functions_in_flambda/. [Accessed 24-02-2025]. July 16, 2021.

[Coc+22] Jesper Cockx et al. "Reasonable Agda is correct Haskell: writing verified Haskell using agda2hs". In: *Haskell '22: 15th ACM SIGPLAN International Haskell Symposium, Ljubljana, Slovenia, September 15 - 16, 2022*. Ed. by Nadia Polikarpova. ACM, 2022, pp. 108–122. ISBN: 978-1-4503-9438-3. DOI: 10.1145/3546189.3549920. URL: https://doi.org/10.1145/3546189.3549920.

[Dyb94] Peter Dybjer. "Inductive Families". In: *Formal Asp. Comput.* 6.4 (1994), pp. 440–465.

[Kle+09] Gerwin Klein et al. "seL4: formal verification of an OS kernel". In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. Ed. by Jeanna Neefe Matthews and Thomas E. Anderson. ACM, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596. URL: http://doi.acm.org/10.1145/1629575.1629596.

[Ler+16] Xavier Leroy et al. "CompCert-a formally verified optimizing compiler". In: *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*. 2016.

[Let02] Pierre Letouzey. "A New Extraction for Coq". In: *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers*. Ed. by Herman Geuvers and Freek Wiedijk. Vol. 2646. Lecture Notes in Computer Science. Springer, 2002, pp. 200–219. ISBN: 3-540-14031-X. URL: http://link.springer.de/link/service/series/0558/bibs/2646/26460200.htm.

[McB10] Conor McBride. "Ornamental algebras, algebraic ornaments". In: *Journal of functional programming* 47 (2010).

[MM04] Conor McBride and James McKinna. "The view from the left". In: *Journal of Functional Programming* 14.1 (2004), pp. 69–111. DOI: 10.1017/S0956796803004829. URL: http://dx.doi.org/10.1017/S0956796803004829.

[Nor08] Ulf Norell. "Dependently Typed Programming in Agda". In: *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*. Ed. by Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra. Vol. 5832. Lecture Notes in Computer Science. Springer, 2008, pp. 230–266. ISBN: 978-3-642-04651-3. DOI: 10.1007/978-3-642-04652-0_5. URL: http://dx.doi.org/10.1007/978-3-642-04652-0_5.

[Sta25] Zoe Stafford. *Make 'CONS', 'NIL', 'JUST' and 'NOTHING' constructors have uniform names by Z-snails · Pull Request #3486 · idris-lang/Idris2 — github.com*. https://github.com/idris-lang/Idris2/pull/3486. [Accessed 13-08-2025]. 2025.

[TB25] Constantine Theocharis and Edwin Brady. "Custom Representations of Inductive Families". In: *arXiv preprint arXiv:2505.21225* (2025).

[Tej20] Matúš Tejiščák. "Erasure in dependently typed programming". PhD thesis. University of St Andrews, 2020. DOI: 10.17630/sta/677. URL: https://doi.org/10.17630/sta/677.

[Wad87] Philip Wadler. "Views: A Way for Pattern Matching to Cohabit with Data Abstraction". In: *POPL*. 1987, pp. 307–313.

[Win24] Théo Winterhalter. "Dependent Ghosts Have a Reflection for Free". In: *Proceedings of the ACM on Programming Languages* 8.ICFP (2024), pp. 630–658. DOI: 10.1145/3674647. URL: https://doi.org/10.1145/3674647.

Acronyms

runid runtime identity

RTT Runid Type Theory

FP Functional Programming

QTT Quantitative Type Theory

BTT Base Type Theory

MLTT Martin-Löf Type Theory

NbE Normalization by Evaluation

PER partial equivalence relation

Appendix A

BTT eliminator typing rules

$$\frac{\Gamma \vdash f \overset{\sigma}{:} (x \overset{\pi}{:} A) \to B \quad \Gamma \vdash a \overset{\sigma\pi}{:} A}{\Gamma \vdash f \overset{\pi}{\cdot} a \overset{\sigma}{:} B} \vdash \cdot$$

$$\frac{\Gamma \vdash c \overset{\sigma}{:} (x \overset{\pi}{:} A) \times B^{\rho} \quad \Gamma \vdash P \overset{0}{:} (y \overset{\sigma}{:} A) \to \mathtt{Set} \quad \Gamma, x \overset{\pi}{:} A, y \overset{\rho}{:} B \vdash b \overset{\sigma}{:} P \overset{\sigma}{\cdot} {}^{\pi}(x,y)^{\rho}}{\Gamma \vdash \mathtt{el}^{\pi} \times^{\rho} c\, P\, b \overset{\sigma}{:} P \overset{\sigma}{\cdot} c} \vdash \mathtt{el} \times$$

$$\frac{\begin{array}{c}\Gamma \vdash n \overset{\sigma}{:} \mathtt{Nat} \qquad \Gamma \vdash P \overset{0}{:} (x \overset{\sigma}{:} \mathtt{Nat}) \to \mathtt{Set} \\ \Gamma \vdash b_z \overset{\sigma}{:} P \overset{\sigma}{\cdot} \mathtt{z} \quad \Gamma, m \overset{\sigma}{:} \mathtt{Nat}, p \overset{\sigma}{:} P \overset{\sigma}{\cdot} m \vdash b_s \overset{\sigma}{:} P \overset{\sigma}{\cdot} (\mathtt{suc}\ m)\end{array}}{\Gamma \vdash \mathtt{elNat}\, n\, P\, b_z\, b_s \overset{\sigma}{:} P \overset{\sigma}{\cdot} n} \vdash \mathtt{elNat}$$

$$\frac{\begin{array}{c}\Gamma \vdash as \overset{\sigma}{:} \mathtt{List}\ A \quad \Gamma \vdash P \overset{0}{:} (y \overset{\sigma}{:} \mathtt{List}\ A) \to \mathtt{Set} \quad \Gamma \vdash b_n \overset{\sigma}{:} P \overset{\sigma}{\cdot} []_l \\ \Gamma, x \overset{\sigma}{:} A, xs \overset{\sigma}{:} \mathtt{List}\ A, p \overset{\sigma}{:} P \overset{\sigma}{\cdot} xs \vdash b_c \overset{\sigma}{:} P \overset{\sigma}{\cdot} (\mathtt{cons}_l\, x\, xs)\end{array}}{\Gamma \vdash \mathtt{elList}\, as\, P\, b_n\, b_c \overset{\sigma}{:} P \overset{\sigma}{\cdot} as} \vdash \mathtt{elList}$$

$$\frac{\begin{array}{c}\Gamma \vdash a \overset{\sigma}{:} \mathtt{Vec}\, A\, n^{\pi} \quad \Gamma \vdash P \overset{0}{:} (m \overset{\pi}{:} \mathtt{Nat}) \to (y \overset{\sigma}{:} \mathtt{Vec}\, A\, m^{\pi}) \to \mathtt{Set} \quad \Gamma \vdash b_n \overset{\sigma}{:} P \overset{\sigma}{\cdot} []_v^{\pi} \\ \Gamma, m \overset{\pi}{:} \mathtt{Nat}, h \overset{\sigma}{:} A, t \overset{\sigma}{:} \mathtt{Vec}\, A\, m^{\pi} \vdash b_c \overset{\sigma}{:} P \overset{\sigma}{\cdot} (\mathtt{cons}_v^{\pi}\, h\, m\, t)\end{array}}{\Gamma \vdash \mathtt{elVec}^{\pi}\, a\, P\, b_n\, b_c \overset{\sigma}{:} P \overset{\sigma}{\cdot} a} \vdash \mathtt{elVec}$$

$$\frac{\begin{array}{cc}\Gamma \vdash s \overset{\sigma}{:} A\, {}^{\pi}\!\uplus^{\rho} B & \Gamma \vdash P \overset{0}{:} (x \overset{\sigma}{:} A\, {}^{\pi}\!\uplus^{\rho} B) \to \mathtt{Set} \\ \Gamma, x \overset{\pi}{:} A \vdash b_L \overset{\sigma\pi}{:} P \overset{\sigma}{\cdot} (\mathtt{inl}^{\pi,\rho} a) & \Gamma, x \overset{\rho}{:} B \vdash b_R \overset{\sigma\rho}{:} P \overset{\sigma}{\cdot} (\mathtt{inr}^{\pi,\rho} b)\end{array}}{\Gamma \vdash \mathtt{el}^{\pi}\, \uplus^{\rho}\, s\, P\, b_L\, b_R \overset{\sigma}{:} P \overset{\sigma}{\cdot} s} \vdash \mathtt{el} \uplus$$

Figure A.1: Typing rules for eliminators

Appendix B

Runid eliminator typing rules

$$\frac{\Gamma_1, \Gamma_2, x \overset{\pi}{:} A, y \overset{\rho}{:} B \vdash b[c \mapsto {}^{\pi}(x,y)^{\rho}] \ \sim_r \ {}^{\pi}(x,y)^{\rho}}{\Gamma_1, c \overset{\omega}{:} (x \overset{\pi}{:} A) \times B^{\rho}, \Gamma_2 \vdash \mathtt{el}_r \, {}^{\pi}\times^{\rho} \, c \, P \, b \overset{\omega}{:} P \overset{\omega}{\cdot} c} \vdash \mathtt{el}_r \times$$

$$\frac{\begin{array}{c}\Gamma_1, \Gamma_2 \vdash b_z[x \mapsto \mathtt{z}] \ \sim_r \ \mathtt{z} \\ \Gamma_1, \Gamma_2, m \overset{\omega}{:} \mathtt{Nat} \vdash b_s[p \mapsto m][x \mapsto \mathtt{suc} \ m] \ \sim_r \ \mathtt{suc} \ m\end{array}}{\Gamma_1, x \overset{\omega}{:} \mathtt{Nat}, \Gamma_2 \vdash \mathtt{el}_r\mathtt{Nat} \, x \, P \, b_z \, b_s \overset{\omega}{:} P \overset{\omega}{\cdot} x} \vdash \mathtt{el}_r\mathtt{Nat}$$

$$\frac{\begin{array}{c}\Gamma_1, \Gamma_2 \vdash b_n[l \mapsto []_l] \ \sim_r \ []_l \\ \Gamma_1, \Gamma_2, h \overset{\omega}{:} A, t \overset{\omega}{:} \mathtt{List} \ A \vdash b_n[p \mapsto t][l \mapsto \mathtt{cons}_l \, h \, t] \ \sim_r \ \mathtt{cons}_l \, h \, t\end{array}}{\Gamma_1, l \overset{\omega}{:} \mathtt{List} \ A, \Gamma_2 \vdash \mathtt{elList} \, l \, A \, P \, b_n \, b_c \overset{\omega}{:} P \overset{\omega}{\cdot} as} \vdash \mathtt{el}_r\mathtt{List}$$

$$\frac{\begin{array}{c}\Gamma_1, \Gamma_2 \vdash b_n[v \mapsto []_v^{\pi}] \ \sim_r \ []_v^{\pi} \\ \begin{pmatrix}\Gamma_1, \Gamma_2, \\ n \overset{\pi}{:} \mathtt{Nat}, \\ h \overset{\omega}{:} A, \\ t \overset{\omega}{:} \mathtt{Vec} \, A \, n^{\pi}\end{pmatrix} \vdash b_c[p \mapsto t][v \mapsto \mathtt{cons}_v^{\pi} \, h \, n \, t] \sim_r \mathtt{cons}_v^{\pi} \, h \, n \, t\end{array}}{\Gamma_1, v \overset{\omega}{:} \mathtt{Vec} \, A \, n^{\pi}, \Gamma_2 \vdash \mathtt{el}_r\mathtt{Vec}^{\pi} \, v \, P \, b_n \, b_c \overset{\omega}{:} P \overset{\omega}{\cdot} v} \vdash \mathtt{el}_r\mathtt{Vec}$$

$$\frac{\Gamma_1, \Gamma_2, y \overset{\omega}{:} B \vdash b_R[s \mapsto \mathtt{inr}^{0,\omega} y] \ \sim_r \ \mathtt{inr}^{0,\omega} y}{\Gamma_1, s \overset{\omega}{:} A \, {}^{0}\uplus^{\omega} B, \Gamma_2 \vdash \mathtt{el}_r \, {}^{0}\uplus^{\omega} \, s \, P \, b_L \, b_R \overset{\omega}{:} P \overset{\omega}{\cdot} s} \vdash \mathtt{el}_r \, {}^{0}\uplus^{\omega}$$

$$\frac{\Gamma_1, \Gamma_2, x \overset{\omega}{:} A \vdash b_L[s \mapsto \mathtt{inl}^{\omega,0} x] \ \sim_r \ \mathtt{inl}^{\omega,0} x}{\Gamma_1, s \overset{\omega}{:} A \, {}^{\omega}\uplus^{0} B, \Gamma_2 \vdash \mathtt{el}_r \, {}^{\omega}\uplus^{0} \, s \, P \, b_L \, b_R \overset{\omega}{:} P \overset{\omega}{\cdot} s} \vdash \mathtt{el}_r \, {}^{\omega}\uplus^{0}$$

$$\frac{\begin{array}{c}\Gamma_1, \Gamma_2, x \overset{\omega}{:} A \vdash b_L[s \mapsto \mathtt{inl}^{\omega,\omega} x] \ \sim_r \ \mathtt{inl}^{\omega,\omega} x \\ \Gamma_1, \Gamma_2, y \overset{\omega}{:} B \vdash b_R[s \mapsto \mathtt{inr}^{\omega,\omega} y] \ \sim_r \ \mathtt{inr}^{\omega,\omega} y\end{array}}{\Gamma_1, s \overset{\omega}{:} A \, {}^{\omega}\uplus^{\omega} B, \Gamma_2 \vdash \mathtt{el}_r \, {}^{\omega}\uplus^{\omega} \, s \, P \, b_L \, b_R \overset{\omega}{:} P \overset{\omega}{\cdot} s} \vdash \mathtt{el}_r \, {}^{\omega}\uplus^{\omega}$$

Figure B.1: Runid Typing rules elims

$$\frac{\Gamma \vdash f \ \sim_r \ f'}{\Gamma \vdash f \overset{0}{\cdot} a \ \sim_r \ f'} \sim_r \overset{0}{\cdot}$$

$$\frac{\Gamma \vdash b_n \ \sim_r \ b'_n \qquad \Gamma, n \overset{0}{:} \mathtt{Nat}, h : A, t : \mathtt{Vec}\, A\, n^0 \vdash b_c \ \sim_r \ b'_c}{\Gamma \vdash \mathtt{elVec}^0\, a\, P\, b_n\, b_c \ \sim_r \ \mathtt{elList}\, a\, P\, b'_n\, b'_c} \sim_r \mathtt{elVec}^0$$

$$\frac{\Gamma, x : A \vdash b \ \sim_r \ c}{\Gamma \vdash \mathtt{el} \times^0 a\, P\, b \ \sim_r \ \lambda(x : A).c \cdot a} \sim_r \mathtt{el} \times^0$$

$$\frac{\Gamma, x \overset{0}{:} A, y : B \vdash b \ \sim_r \ c}{\Gamma \vdash \mathtt{el}^0 \times a\, P\, b \ \sim_r \ \lambda(y : B).c \cdot a} \sim_r \mathtt{el}^0 \times$$

$$\frac{\Gamma, x : A \vdash b_L \ \sim_r \ c}{\Gamma \vdash \mathtt{el} \uplus^0 a\, P\, b_L\, b_R \ \sim_r \ \lambda(x : A).c \cdot a} \sim_r \mathtt{el} \uplus^0$$

$$\frac{\Gamma, x : B \vdash b_R \ \sim_r \ c}{\Gamma \vdash \mathtt{el}^0 \uplus a\, P\, b_L\, b_R \ \sim_r \ \lambda(x : B).c \cdot a} \sim_r \mathtt{el}^0 \uplus$$

Figure B.2: Eliminator erasure rules

Appendix C

Erasure function

(Types)

$$
\begin{array}{ll}
\downarrow (a \overset{0}{:} A) \to B & = \downarrow B \\
\downarrow (a \overset{\omega}{:} A) \to B & = (a :\downarrow A) \to \downarrow B \\
\downarrow (a \overset{\omega}{:} A) \times B^{\omega} & = (\downarrow a :\downarrow A) \times \downarrow B \\
\downarrow (a \overset{\omega}{:} A) \times B^{0} & = \downarrow A \\
\downarrow (a \overset{0}{:} A) \times B^{\omega} & = \downarrow B \\
\downarrow \texttt{List}\ A & = \texttt{List}\ \downarrow A \\
\downarrow \texttt{Vec}\, A\, n^{0} & = \texttt{List}\ \downarrow A \\
\downarrow \texttt{Vec}\, A\, n^{\omega} & = \texttt{Vec}\ \downarrow A\ \downarrow n \\
\downarrow A\, {}^{\omega}\!\uplus^{\omega} B & = \downarrow A \uplus \downarrow B \\
\downarrow A\, {}^{0}\!\uplus^{\omega} B & = \downarrow B \\
\downarrow A\, {}^{\omega}\!\uplus^{0} B & = \downarrow A \\
\downarrow \texttt{Nat} & = \texttt{Nat}
\end{array}
$$

(Constructors)

$$
\begin{array}{ll}
\downarrow \lambda(a \overset{0}{:} A).b & = \downarrow b \\
\downarrow \lambda(a \overset{\omega}{:} A).b & = \lambda(a :\downarrow A).\ \downarrow b \\
\downarrow {}^{\omega}(a, b)^{\omega} & = (\downarrow a, \downarrow b) \\
\downarrow {}^{0}(a, b)^{\omega} & = \downarrow b \\
\downarrow {}^{\omega}(a, b)^{0} & = \downarrow a \\
\downarrow []_{l} & = []_{l} \\
\downarrow \texttt{cons}_{l}\, h\, t & = \texttt{cons}_{l}\ \downarrow h\ \downarrow t \\
\downarrow []_{v}^{0} & = []_{l} \\
\downarrow []_{v}^{\omega} & = []_{v} \\
\downarrow \texttt{cons}_{v}^{0}\, h\, n\, t & = \texttt{cons}_{l}\ \downarrow h\ \downarrow t \\
\downarrow \texttt{cons}_{v}^{\omega}\, h\, n\, t & = \texttt{cons}_{v}\ \downarrow h\ \downarrow n\ \downarrow t \\
\downarrow \texttt{inl}^{\omega,\omega} a & = \texttt{inl}^{\omega,\omega}\ \downarrow a \\
\downarrow \texttt{inl}^{\omega,0} a & = \downarrow a \\
\downarrow \texttt{inr}^{\omega,\omega} b & = \texttt{inr}^{\omega,\omega}\ \downarrow b \\
\downarrow \texttt{inr}^{0,\omega} b & = \downarrow b \\
\downarrow \texttt{z} & = \texttt{z} \\
\downarrow \texttt{suc}\ n & = \texttt{suc}\ \downarrow n
\end{array}
$$

(Eliminators)

$$
\begin{aligned}
&\downarrow f \overset{0}{\cdot} a && = \downarrow f \\
&\downarrow f \overset{\omega}{\cdot} a && = \downarrow f \cdot \downarrow a \\
&\downarrow \mathtt{el}^{\omega} \times^{\omega} a\, P\, b && = \mathtt{el} \times \downarrow a \downarrow P \downarrow b \\
&\downarrow \mathtt{el}^{0} \times^{\omega} a\, P\, b && = (\lambda(y :\downarrow B).\downarrow b) \cdot \downarrow a \\
&\downarrow \mathtt{el}^{\omega} \times^{0} a\, P\, b && = (\lambda(x :\downarrow A).\downarrow b) \cdot \downarrow a \\
&\downarrow \mathtt{el}_r{}^{\pi} \times^{\rho} a\, P\, b && = \downarrow \mathtt{el}^{\pi} \times^{\rho} a\, P\, b \\
&\downarrow \mathtt{elList}\, a\, P\, b\, c && = \mathtt{elList} \downarrow a \downarrow P \downarrow b \downarrow c \\
&\downarrow \mathtt{el}_r\mathtt{List}\, a\, P\, b\, c && = \downarrow \mathtt{elList}\, a\, P\, b\, c \\
&\downarrow \mathtt{elVec}^{\omega}\, a\, P\, b\, c && = \mathtt{elVec} \downarrow a \downarrow P \downarrow b \downarrow c \\
&\downarrow \mathtt{elVec}^{0}\, a\, P\, b\, c && = \mathtt{elList} \downarrow a \downarrow P \downarrow b \downarrow c \\
&\downarrow \mathtt{el}_r\mathtt{Vec}^{\pi}\, a\, P\, b\, c && = \downarrow \mathtt{elVec}^{\pi}\, a\, P\, b\, c \\
&\downarrow \mathtt{el}^{\omega} \uplus^{\omega} s\, P\, b\, c && = \mathtt{el} \uplus \downarrow s \downarrow P \downarrow b \downarrow c \\
&\downarrow \mathtt{el}^{0} \uplus^{\omega} s\, P\, b\, c && = (\lambda(x :\downarrow B).\downarrow c) \cdot \downarrow s \\
&\downarrow \mathtt{el}^{\omega} \uplus^{0} s\, P\, b\, c && = (\lambda(x :\downarrow A).\downarrow b) \cdot \downarrow s \\
&\downarrow \mathtt{el}_r{}^{\pi} \uplus^{\rho} s\, P\, b\, c && = \downarrow \mathtt{el}^{\pi} \uplus^{\rho} s\, P\, b\, c \\
&\downarrow \mathtt{elNat}\, a\, P\, b\, c && = \mathtt{elNat} \downarrow a \downarrow P \downarrow b \downarrow c \\
&\downarrow \mathtt{el}_r\mathtt{Nat}\, a\, P\, b\, c && = \downarrow \mathtt{elNat}\, a\, P\, b\, c
\end{aligned}
$$

Appendix D

Strong runtime equivalence relation

D.1 Types

$$\frac{\Gamma, a \overset{0}{:} A \vdash B \sim_s D : \mathtt{Set}}{\Gamma \vdash (a \overset{0}{:} A) \to B \sim_s D : \mathtt{Set}} \qquad \frac{\Gamma \vdash A \sim_s C : \mathtt{Set} \quad \Gamma, a \overset{\omega}{:} A \vdash B \sim_s D : \mathtt{Set}}{\Gamma \vdash (a \overset{\omega}{:} A) \to B \sim_s (c \overset{\omega}{:} C) \to D : \mathtt{Set}}$$

$$\frac{\Gamma \vdash A \sim_s A : \mathtt{Set} \quad \Gamma, a \overset{\omega}{:} A \vdash B \sim_s A : \mathtt{Set}}{\Gamma \vdash (a : A) \to_r B \sim_s (a : A) \to_r A : \mathtt{Set}}$$

Figure D.1: Strong relation on function types

$$\frac{\Gamma, x \overset{0}{:} A \vdash B \sim_s C : \mathtt{Set}}{\Gamma \vdash (x \overset{0}{:} A) \times B^{\omega} \sim_s C : \mathtt{Set}} \qquad \frac{\Gamma \vdash A \sim_s C : \mathtt{Set}}{\Gamma \vdash (x \overset{\omega}{:} A) \times B^{0} \sim_s C : \mathtt{Set}}$$

$$\frac{\Gamma \vdash A \sim_s C : \mathtt{Set} \quad \Gamma, x \overset{\omega}{:} A \vdash B \sim_s D : \mathtt{Set}}{\Gamma \vdash (x \overset{\omega}{:} A) \times B^{\omega} \sim_s (x \overset{\omega}{:} C) \times D^{\omega} : \mathtt{Set}}$$

Figure D.2: Strong relation on product types

$$\frac{\Gamma \vdash A \sim_s B : \mathtt{Set}}{\Gamma \vdash \mathtt{List}\ A \sim_s \mathtt{List}\ B : \mathtt{Set}}$$

Figure D.3: Strong relation on list type

$$\frac{\Gamma \vdash A \sim_s B : \mathtt{Set}}{\Gamma \vdash \mathtt{Vec}\, A\, n^{0} \sim_s \mathtt{List}\ B : \mathtt{Set}} \qquad \frac{\Gamma \vdash A \sim_s B : \mathtt{Set} \quad \Gamma \vdash n \sim_s m : \mathtt{Nat}}{\Gamma \vdash \mathtt{Vec}\, A\, n^{\omega} \sim_s \mathtt{Vec}\, B\, m^{\omega} : \mathtt{Set}}$$

Figure D.4: Strong relation on vector type

$$\frac{}{\Gamma \vdash \texttt{Nat} \sim_s \texttt{Nat} : \texttt{Set}}$$

Figure D.5: Strong relation on nat type

$$\frac{\Gamma \vdash B \sim_s B : \texttt{Set}}{\Gamma \vdash A\,{}^{0}\!\uplus^{\omega} B \sim_s B : \texttt{Set}} \qquad \frac{\Gamma \vdash A \sim_s A : \texttt{Set}}{\Gamma \vdash A\,{}^{\omega}\!\uplus^{0} B \sim_s A : \texttt{Set}}$$

$$\frac{\Gamma \vdash A \sim_s C : \texttt{Set} \quad \Gamma \vdash B \sim_s D : \texttt{Set}}{\Gamma \vdash A\,{}^{\omega}\!\uplus^{\omega} B \sim_s C\,{}^{\omega}\!\uplus^{\omega} D : \texttt{Set}}$$

Figure D.6: Strong relation on sum types

D.2 Terms

D.2.1 Constructors

$$\frac{\Gamma, a \overset{0}{:} A \vdash b \sim_s c : B}{\Gamma \vdash \lambda(a \overset{0}{:} A).b \sim_s c : B} \qquad \frac{\Gamma, a \overset{\omega}{:} A \vdash b \sim_s b' : B}{\Gamma \vdash \lambda(a \overset{\omega}{:} A).b \sim_s \lambda(a \overset{\omega}{:} A).b' : (a \overset{\omega}{:} A) \to B}$$

$$\frac{\Gamma, a \overset{\omega}{:} A \vdash b \sim_s a : A}{\Gamma \vdash \lambda_r(a : A).b \sim_s \lambda(a : A).a : (a : A) \to_r A}$$

Figure D.7: Strong relation on lambdas

$$\frac{\Gamma, x \overset{0}{:} A \vdash b \sim_s c : C}{\Gamma \vdash {}^{0}(a, b)^{\omega} \sim_s c : C} \qquad \frac{\Gamma \vdash a \sim_s a : A}{\Gamma \vdash {}^{\omega}(a, b)^{0} \sim_s a : A}$$

$$\frac{\Gamma \vdash a \sim_s c : A \quad \Gamma, x \overset{\omega}{:} A \vdash b \sim_s d : B}{\Gamma \vdash {}^{\omega}(a, b)^{\omega} \sim_s {}^{\omega}(c, d)^{\omega} : (x \overset{\omega}{:} A) \times B^{\omega}}$$

Figure D.8: Strong relation on pairs

$$\frac{}{\Gamma \vdash \mathtt{z} \sim_s \mathtt{z} : \mathtt{Nat}} \qquad \frac{\Gamma \vdash n \sim_s m : \mathtt{Nat}}{\Gamma \vdash \mathtt{suc}\ n \sim_s \mathtt{suc}\ m : \mathtt{Nat}}$$

Figure D.9: Strong relation on `Nat` constructors

$$\frac{}{\Gamma \vdash []_l \sim_s []_l : \mathtt{List}\ A} \qquad \frac{\Gamma \vdash h \sim_s h' : A \quad \Gamma \vdash t \sim_s t' : \mathtt{List}\ A}{\Gamma \vdash \mathtt{cons}_l\, h\, t \sim_s \mathtt{cons}_l\, h'\, t' : \mathtt{List}\ A}$$

Figure D.10: Strong relation on `List` constructors

$$\frac{}{\Gamma \vdash \texttt{[]}^0_v \sim_s \texttt{[]}_l : \texttt{Vec}\, A\, \texttt{z}^0} \qquad \frac{}{\Gamma \vdash \texttt{[]}^\omega_v \sim_s \texttt{[]}^\omega_v : \texttt{Vec}\, A\, \texttt{z}^\omega}$$

$$\frac{\Gamma \vdash h \sim_s h' : A \quad \Gamma \vdash t \sim_s t' : \texttt{Vec}\, A\, n^0}{\Gamma \vdash \texttt{cons}^0_v\, h\, n\, t \sim_s \texttt{cons}_l\, h'\, t' : \texttt{Vec}\, A\, \texttt{suc}\ \ n^0} \qquad \frac{\Gamma \vdash h \sim_s h' : A \quad \Gamma \vdash t \sim_s t' : \texttt{Vec}\, A\, n^\omega \quad \Gamma \vdash n \sim_s n' : \texttt{Nat}}{\Gamma \vdash \texttt{cons}^\omega_v\, h\, n\, t \sim_s \texttt{cons}^\omega_v\, h'\, n'\, t' : \texttt{Vec}\, A\, \texttt{suc}\ \ n^\omega}$$

Figure D.11: Strong relation on `Vec` constructors

$$\frac{\Gamma \vdash a \sim_s a' : A}{\Gamma \vdash \texttt{inl}^{\omega,\omega} a \sim_s \texttt{inl}^{\omega,\omega} a' : A\, {}^\omega\!\uplus^\omega B} \qquad \frac{\Gamma \vdash b \sim_s b' : B}{\Gamma \vdash \texttt{inr}^{\omega,\omega} b \sim_s \texttt{inr}^{\omega,\omega} b' : A\, {}^\omega\!\uplus^\omega B}$$

$$\frac{\Gamma \vdash a \sim_s a' : A}{\Gamma \vdash \texttt{inl}^{\omega,0} a \sim_s a' : A\, {}^\omega\!\uplus^0 B} \qquad \frac{\Gamma \vdash b \sim_s b' : B}{\Gamma \vdash \texttt{inr}^{0,\omega} b \sim_s \texttt{inr}^{0,\omega} b' : A\, {}^0\!\uplus^\omega B}$$

Figure D.12: Strong relation on sum injections

D.2.2 Eliminators

$$\frac{\Gamma \vdash f \sim_s \lambda(x \overset{0}{:} A).b : (x \overset{0}{:} A) \to B}{\Gamma \vdash f \overset{0}{\cdot} a \sim_s b : B} \qquad \frac{\Gamma \vdash f \sim_s f' : (x \overset{\omega}{:} A) \to B \quad \Gamma \vdash a \sim_s a' : a}{\Gamma \vdash f \overset{\omega}{\cdot} a \sim_s f' \overset{\omega}{\cdot} a' : B}$$

$$\frac{\Gamma \vdash f \sim_s \lambda_r(x : A).x : (a \overset{\omega}{:} A) \to A \quad \Gamma \vdash a \sim_s a : A}{\Gamma \vdash f \cdot_r a \sim_s a : A}$$

Figure D.13: Strong relation for function application

$$\frac{\begin{array}{c}\Gamma \vdash a \sim_s a' : \mathtt{Nat} \qquad \Gamma \vdash P \sim_s P' : (n \overset{\omega}{:} \mathtt{Nat}) \to \mathtt{Set} \\ \Gamma \vdash b \sim_s b' : P \cdot \mathtt{z} \quad \Gamma, m \overset{\omega}{:} \mathtt{Nat}, p \overset{\omega}{:} P \cdot m \vdash c \sim_s c' : P \cdot (\mathtt{suc}\ m)\end{array}}{\Gamma \vdash \mathtt{elNat}\, a\, P\, b\, c \sim_s \mathtt{elNat}\, a'\, P'\, b'\, c' : P \cdot a}$$

$$\frac{\begin{array}{c}\Gamma_1, a \overset{\omega}{:} \mathtt{Nat}, \Gamma_2 \vdash a \sim_s a : \mathtt{Nat} \\ \Gamma_1, \Gamma_2 \vdash b[a \mapsto \mathtt{z}] \sim_s \mathtt{z} : \mathtt{Nat} \\ \Gamma_1, \Gamma_2, m \overset{\omega}{:} \mathtt{Nat} \vdash c[p \mapsto m][a \mapsto \mathtt{suc}\ m] \sim_s \mathtt{suc}\ m : \mathtt{Nat} \\ \Gamma_1, a \overset{\omega}{:} \mathtt{Nat}, \Gamma_2, m \overset{\omega}{:} \mathtt{Nat}, p \overset{\omega}{:} \mathtt{Nat} \vdash c \sim_s c : \mathtt{Nat}\end{array}}{\Gamma_1, a \overset{\omega}{:} \mathtt{Nat}, \Gamma_2 \vdash \mathtt{el}_r\mathtt{Nat}\, a\, P\, b\, c \sim_s a : \mathtt{Nat}}$$

Figure D.14: Strong relation for `Nat` eliminator

$$\frac{\begin{array}{c}\Gamma \vdash p \sim_s p : A \quad \Gamma \vdash P \sim_s P : (z \overset{\omega}{:} (x \overset{0}{:} A) \times B^{\omega}) \to \mathtt{Set} \\ \Gamma, x \overset{0}{:} A, y \overset{\omega}{:} B \vdash b \sim_s c : P \cdot y\end{array}}{\Gamma \vdash \mathtt{el}^{\,0} \times^{\omega} p\, P\, b \sim_s \lambda(a \overset{\omega}{:} B).c \overset{\omega}{\cdot} p : P \cdot a}$$

$$\frac{\begin{array}{c}\Gamma \vdash p \sim_s p : A \quad \Gamma \vdash P \sim_s P : (z \overset{\omega}{:} (x \overset{\omega}{:} A) \times B^{0}) \to \mathtt{Set} \\ \Gamma, x \overset{\omega}{:} A, y \overset{0}{:} B \vdash b \sim_s c : P \cdot x\end{array}}{\Gamma \vdash \mathtt{el}^{\,\omega} \times^{0} p\, P\, b \sim_s \lambda(a \overset{\omega}{:} A).c \overset{\omega}{\cdot} p : P \cdot a}$$

$$\frac{\begin{array}{c}\Gamma \vdash p \sim_s p' : (a \overset{\omega}{:} A) \times B^{\omega} \quad \Gamma \vdash P \sim_s P' : (z \overset{\omega}{:} (a \overset{\omega}{:} A) \times B^{\omega}) \to \mathtt{Set} \\ \Gamma, x \overset{\omega}{:} A, y \overset{\omega}{:} B \vdash b \sim_s b' : (a \overset{\omega}{:} A) \times B^{\omega}\end{array}}{\Gamma \vdash \mathtt{el}^{\,\omega} \times^{\omega} p\, P\, b \sim_s \mathtt{el}^{\,\omega} \times^{\omega} p'\, P'\, b' : (a \overset{\omega}{:} A) \times B^{\omega}}$$

$$\frac{\begin{array}{c}\Gamma \vdash p \sim_s p : (a \overset{\pi}{:} A) \times B^{\rho} \\ \Gamma, x \overset{\pi}{:} A, y \overset{\rho}{:} B \vdash b \sim_s {}^{\pi}(x, y)^{\rho} : (a \overset{\pi}{:} A) \times B^{\rho}\end{array}}{\Gamma \vdash \mathtt{el}_r{}^{\pi} \times^{\rho} p\_ b \sim_s a : (a \overset{\pi}{:} A) \times B^{\rho}}$$

Figure D.15: Strong relation for $\times$ eliminator

$$\frac{\Gamma \vdash P \sim_s P : (p \overset{\omega}{:} A\,{}^{0}\!\uplus^{\omega} B) \to \mathtt{Set} \quad \Gamma \vdash s \sim_s s : B \quad \Gamma, x \overset{\omega}{:} B \vdash b_L \sim_s b : P \overset{\omega}{\cdot} (\mathtt{inr}^{0,\omega} x)}{\Gamma \vdash \mathtt{el}\,{}^{0}\!\uplus^{\omega} s\, P\, b_L\, b_R \sim_s (\lambda(x \overset{\omega}{:} B).b) \overset{\omega}{\cdot} s : P \overset{\omega}{\cdot} s}$$

$$\frac{\Gamma \vdash P \sim_s P : (p \overset{\omega}{:} A\,{}^{\omega}\!\uplus^{0} B) \to \mathtt{Set} \quad \Gamma \vdash s \sim_s s : A \quad \Gamma, x \overset{\omega}{:} A \vdash b_R \sim_s b : P \overset{\omega}{\cdot} (\mathtt{inl}^{\omega,0} x)}{\Gamma \vdash \mathtt{el}\,{}^{\omega}\!\uplus^{0} s\, P\, b_L\, b_R \sim_s (\lambda(x \overset{\omega}{:} A).b) \overset{\omega}{\cdot} s : P \overset{\omega}{\cdot} s}$$

$$\frac{\begin{array}{c}\Gamma \vdash P \sim_s P' : (p \overset{\omega}{:} A\,{}^{\omega}\!\uplus^{\omega} B) \to \mathtt{Set} \quad \Gamma \vdash s \sim_s s' : A\,{}^{\omega}\!\uplus^{\omega} B \\ \Gamma, x \overset{\omega}{:} A \vdash b_L \sim_s b'_L : P \overset{\omega}{\cdot} (\mathtt{inl}^{\omega,\omega} x) \\ \Gamma, y \overset{\omega}{:} B \vdash b_R \sim_s b'_R : P \overset{\omega}{\cdot} (\mathtt{inr}^{\omega,\omega} y)\end{array}}{\Gamma \vdash \mathtt{el}\,{}^{\omega}\!\uplus^{\omega} s\, P\, b_L\, b_R \sim_s \mathtt{el}\,{}^{\omega}\!\uplus^{\omega} s'\, P'\, b'_L\, b'_R : P \overset{\omega}{\cdot} s}$$

$$\frac{\Gamma_1, s \overset{\omega}{:} B, \Gamma_2 \vdash P \sim_s P : (x \overset{\omega}{:} B) \to \mathtt{Set} \quad \Gamma_1, \Gamma_2, x \overset{\omega}{:} B \vdash b_R[s \mapsto x] \sim_s x : B}{\Gamma_1, s \overset{\omega}{:} B, \Gamma_2 \vdash \mathtt{el}_r\,{}^{0}\!\uplus^{\omega} s\, P\, b_L\, b_R \sim_s s : B}$$

$$\frac{\Gamma_1, s \overset{\omega}{:} A, \Gamma_2 \vdash P \sim_s P : (x \overset{\omega}{:} A) \to \mathtt{Set} \quad \Gamma_1, \Gamma_2, x \overset{\omega}{:} A \vdash b_L[s \mapsto x] \sim_s x : A}{\Gamma_1, s \overset{\omega}{:} A, \Gamma_2 \vdash \mathtt{el}_r\,{}^{\omega}\!\uplus^{0} s\, P\, b_L\, b_R \sim_s s : A}$$

$$\frac{\begin{array}{c}\Gamma_1, s \overset{\omega}{:} A\,{}^{\omega}\!\uplus^{\omega} B, \Gamma_2 \vdash P \sim_s P : (x \overset{\omega}{:} A\,{}^{\omega}\!\uplus^{\omega} B) \to \mathtt{Set} \\ \Gamma_1, \Gamma_2, x \overset{\omega}{:} A \vdash b_L[s \mapsto \mathtt{inl}^{\omega,\omega} x] \sim_s \mathtt{inl}^{\omega,\omega} x : A\,{}^{\omega}\!\uplus^{\omega} B \\ \Gamma_1, \Gamma_2, y \overset{\omega}{:} B \vdash b_R[s \mapsto \mathtt{inr}^{\omega,\omega} y] \sim_s \mathtt{inr}^{\omega,\omega} y : A\,{}^{\omega}\!\uplus^{\omega} B\end{array}}{\Gamma_1, s \overset{\omega}{:} A\,{}^{\omega}\!\uplus^{\omega} B, \Gamma_2 \vdash \mathtt{el}_r\,{}^{\omega}\!\uplus^{\omega} s\, P\, b_L\, b_R \sim_s s : A\,{}^{\omega}\!\uplus^{\omega} B}$$

Figure D.16: Strong relation for $\uplus$ eliminators

$$\frac{\begin{array}{c}\Gamma \vdash a \sim_s a' : \mathtt{List}\ A \quad \Gamma \vdash P \sim_s P' : (x \overset{\omega}{:} \mathtt{List}\ A) \to \mathtt{Set} \quad \Gamma \vdash b \sim_s b' : P \overset{\omega}{\cdot} []_l \\ \Gamma, h \overset{\omega}{:} A, t \overset{\omega}{:} \mathtt{List}\ A, p \overset{\omega}{:} P \overset{\omega}{\cdot} t \vdash c \sim_s c' : P \overset{\omega}{\cdot} (\mathtt{cons}_l\, h\, t)\end{array}}{\Gamma \vdash \mathtt{elList}\, a\, P\, b\, c \sim_s \mathtt{elList}\, a'\, P'\, b'\, c' : P \overset{\omega}{\cdot} a}$$

$$\frac{\begin{array}{c}\Gamma_1, a \overset{\omega}{:} \mathtt{List}\ A, \Gamma_2 \vdash P \sim_s P : (x \overset{\omega}{:} \mathtt{List}\ A) \to \mathtt{Set} \\ \Gamma_1, \Gamma_2 \vdash b[a \mapsto []_l] \sim_s []_l : \mathtt{List}\ A \\ \Gamma_1, \Gamma_2, h \overset{\omega}{:} A, t \overset{\omega}{:} \mathtt{List}\ A \vdash c[p \mapsto t][a \mapsto \mathtt{cons}_l\, h\, t] \sim_s \mathtt{cons}_l\, h\, t : \mathtt{List}\ A\end{array}}{\Gamma_1, a \overset{\omega}{:} \mathtt{List}\ A, \Gamma_2 \vdash \mathtt{el}_r\mathtt{List}\, a\, P\, b\, c \sim_s a : \mathtt{List}\ A}$$

Figure D.17: Strong relation for $\mathtt{List}$ eliminators

$$\frac{\Gamma \vdash a \sim_s a' : \texttt{List}\ A \qquad \Gamma \vdash b \sim_s b' : P \overset{\omega}{\cdot} \texttt{[]}_l \qquad \Gamma, h \overset{\omega}{:} A, n \overset{0}{:} \texttt{Nat}, t \overset{\omega}{:} \texttt{List}\ A, p \overset{\omega}{:} P \overset{\omega}{\cdot} t \vdash c \sim_s c' : P \overset{\omega}{\cdot} \texttt{cons}_l\, h\, t}{\Gamma \vdash \texttt{elVec}^0\, a\, P\, b\, c \sim_s \texttt{elList}\, a'\, P'\, b'\, c' : P \overset{\omega}{\cdot} a}$$

$$\frac{\Gamma \vdash a \sim_s a' : \texttt{Vec}\, A\, n^\omega \qquad \Gamma \vdash b \sim_s b' : P \overset{\omega}{\cdot} \texttt{[]}_v^\omega \qquad \Gamma, h \overset{\omega}{:} A, m \overset{\omega}{:} \texttt{Nat}, t \overset{\omega}{:} \texttt{Vec}\, A\, m^\omega, p \overset{\omega}{:} P \overset{\omega}{\cdot} t \vdash c \sim_s c' : P \overset{\omega}{\cdot} \texttt{cons}_v^\omega\, h\, m\, t}{\Gamma \vdash \texttt{elVec}^\omega\, a\, P\, b\, c \sim_s \texttt{elVec}^\omega\, a'\, P'\, b'\, c' : P \overset{\omega}{\cdot} a}$$

$$\frac{\begin{array}{c}\Gamma_1, a \overset{\omega}{:} \texttt{Vec}\, A\, n^\pi, \Gamma_2 \vdash a \sim_s a : \texttt{Vec}\, A\, n^\pi \\ \Gamma_1, \Gamma_2 \vdash b[a \mapsto \texttt{[]}_v^\pi] \sim_s \texttt{[]}_v^\pi : \texttt{Vec}\, A\, \texttt{z}^\pi \\ \Gamma_1, \Gamma_2, h \overset{\omega}{:} A, m \overset{\pi}{:} \texttt{Nat}, t \overset{\omega}{:} \texttt{Vec}\, A\, m^\pi \vdash c[p \mapsto t][a \mapsto \texttt{cons}_v^\pi\, h\, m\, t] \sim_s \texttt{cons}_v^\pi\, h\, m\, t : \texttt{Vec}\, A\, (\texttt{suc}\ m)^\pi\end{array}}{\Gamma_1, a \overset{\omega}{:} \texttt{Vec}\, A\, n^\pi, \Gamma_2 \vdash \texttt{el}_r\texttt{Vec}^\pi\, a\, \_\, b\, c \sim_s a : \texttt{Vec}\, A\, n^\pi}$$

Figure D.18: Strong relation on vector eliminators