Run-time identity functions in a quantitative type theory

José Carlos Padilla Cancio[1][0009−0006−6787−162X], Jesper Cockx[1][0000−0003−3862−4073], and Bohdan Liesnikov[1][0009−0000−2216−8830]

Delft University of Technology, Delft, The Netherlands
jcpadillacancio@gmail.com, J.G.H.Cockx@tudelft.nl, B.Liesnikov@tudelft.nl

Abstract. Many dependently typed languages impose run-time overhead because they must track proof-relevant information throughout execution. Quantitative type theory (QTT) addresses this by distinguishing compile-time and run-time data. However, this distinction leads to situations where data types differ in their erased proofs while sharing the same run-time representation. Functions operating only on such erased proofs behave as run-time identity functions without the type theory explicitly recognizing them.

We introduce a type theory extending QTT with a principled notion of run-time identity (runid) functions. Our system adds runid annotations to the syntax and equips the language with a syntactic notion of equivalence of terms at run-time. To reason about this syntactic relation we introduce a denotational semantics. We prove that these annotations correctly capture when terms are equivalent at run time, establishing the coherence between the syntactic and semantic accounts.

Keywords: Dependent types · Inductive families · Structural typing.

1 Introduction

Society often pays a high price to mitigate consequences of bugs. Formal verification allows us to prove that software adheres to specification and thus gives strong correctness guarantees. However, verification often comes at a high cost — lower performance of the end program being a frequent one.

Dependent type systems are one established way to formally verify software. To manage the overhead of proofs at run-time, they can include *erasure* [2]; which allows programmers to communicate to the compiler which parts can be safely removed during compilation. As an example, vectors can have their length index marked as erased such as in Listing 1.

For formal proofs to be easier, programmers often have to define multiple data types [10,7] that have same runtime representation, but carry proofs of different invariants. And while erasure allows the programmer to communicate what the type should look like at runtime, it does not reflect this knowledge back into the type system.

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

Listing 1: Defition of vector with erased length index in Agda

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

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

Listing 2: Definition of lists and listToVec in Agda

The `listToVec` function that maps from list to vector, would not do any work at run-time! However, it will still be present and induce a useless traversal of the input. This additional overhead is what we aim to eliminate in this paper.

Existing solutions [4,9] based on ornaments [6] have some limitations. Chiefly, they tend to rely on unergonomic encodings of data types. Programmers have to manually write mappings from every single existing data type into its corresponding ornament (e.g. all list-based types into the base list type). These mappings also tend to imply manual proving of certain conditions that are implicit in the data type. As a concrete example, 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.

For a practical idea of what we aim for, consider the mock in Listing 3. On top of the regular definitions of inductive types and functions on them — we add annotations, informing the type system that we intend `Vec` to be compiled away to a `List` and that `listToVec` should be compiled away completely. The compiler can then use the information gleaned from these annotations to verify their validity and the behaviour of the code. It then uses this newly confirmed information to optimize away these run-time irrelevant details. Since the runid status is an explicit member of the type system the compiler can even propagate this runid status or infer it, such as a map function `@runid map f : List A -> List B` supplied with a runid function `f`.

Contrary to ornament based approaches the programmer would not need to define views as refinements of the base type, nor provide manual coherence proofs. The annotations provide the type system with sufficient information to perform a structural analysis which verifies runtime equivalences and thus validity of runid functions.

```
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 3: Mock data type and runid annotations

For this paper we decide to focus on the type system and assume that inference is being done in a previous pass to the state we receive the intermediate representation in. As such our contributions are:

- Core type theory with exhaustive runid and erasure annotations.
- Syntactic run-time-equivalence relation $\Gamma \vdash a \sim_r b$, used in the type theory.
- Guidance for implementing this as a language feature.
- Denotational semantics of run-time equivalence post-erasure using an extensional semantic domain based on a PER model[1].
- Sketch of syntactic run-time equivalence soundness proof in the style of logical relations: Terms related by our run-time equivalence relation are mapped to equal values in our semantic domain.

2 Extending QTT with runid functions

We extend QTT with a runid marking r. We rely on the Barendregt convention in this paper, thus ignoring variable name capture. In this section we present a calculus with functions $(a \overset{\pi}{:} A) \to B$ and one primitive type `Nat` for brevity. Further types — dependent products, sums, lists and vectors — are elaborated on in appendix A.

2.1 Syntax

Figure 1 showcases the syntax of our simplified calculus. Functions carry a usage annotation π for their argument x in both the type $(x \overset{\pi}{:} A) \to B$, constructor $\lambda(x \overset{\pi}{:} A).b$ and eliminator $a \overset{\pi}{\cdot} b$. Usages can be either erased 0 or present ω.

Since runid functions cannot take erased input when function syntax is marked runid we dont provide a usage annotation. Similar to functions, `Nat` eliminators `elNat` can carry a runid marking. Contexts $\Gamma = \Gamma', x \overset{\sigma}{:} A$ map a variable x to both its type A and usage σ.

$$
\begin{array}{rcll}
x, y, z, p & \in & \texttt{String} & \\
\pi, \sigma, \rho & ::= & 0 \mid \omega & \\
\Gamma & ::= & \emptyset \mid \Gamma, x \overset{\pi}{:} A & \\
A, B & ::= & (x \overset{\pi}{:} A) \to B \mid (x : A) \to_r B \mid \texttt{Nat} \mid \texttt{Set} & \text{Types} \\
a, b, c, P & ::= & x & \text{Variables} \\
& & \mid \lambda(x \overset{\pi}{:} A).b \mid \lambda_r(x : A).b \mid a \overset{\pi}{\cdot} c \mid a \cdot_r b & \text{Functions} \\
& & \mid \texttt{suc}\ \ a \mid \texttt{z} \mid \texttt{elNat}\, a\, P\, b\, c \mid \texttt{el}_r\texttt{Nat}\, a\, P\, b\, c & \texttt{Nat}
\end{array}
$$

Fig. 1. Syntax

2.2 Run-time Equivalence

This section covers the primary contribution of the paper: the weak run-time equivalence relation $\Gamma \vdash a \ \sim_r \ b$. This relation allows us to state syntactically that two terms have the same value or behaviour at run-time, which we make use of in our typing rules for runid terms.

The majority of the rules are not surprising: the run-time equivalence relation shares the same congruence and equivalence rules present in conversion (but no computation rules). The interesting rules concern terms annotated with erasure or marked runid. The terms with usage annotations or runid markings are related to the "optimized" version — with identity computations removed.

Relating runid terms We call it a weak relation because it only needs to to be defined on well-typed terms. Since it is only used in the well-typedness judgment for runid terms, any ill-formed terms would be rejected by the precondition that any runid term must be a well-typed regular term.

Figure 2 applies this to functions and eliminators by equating them to their optimizations. Therefore, runid lambdas are equated to a trivial identity function and runid eliminators are equated to their argument.

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

$$
\frac{}{\Gamma \vdash \lambda_r(a : A).b \ \sim_r \ \lambda(a : A).b} \sim_r \texttt{LAMR} \quad \frac{}{\Gamma \vdash f \cdot_r a \ \sim_r \ a} \sim_r \texttt{APPR}
$$

$$
\frac{}{\Gamma \vdash \texttt{el}_r\texttt{Nat}\, x\, P\, b\, c \ \sim_r \ x} \sim_r \texttt{ELNATR}
$$

Fig. 2. Relating runid terms

Relating types with erasure Figure 3 showcases the rules for types. Dependent function types weaken the codomain of the result type with the argument variable $\Gamma, x \overset{0}{:} A \vdash B \overset{0}{:} \texttt{Set}$. 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.

If two types are considered to be equivalent at run-time $\Gamma \vdash A \ \sim_r \ B$ then for each instance $\Gamma \vdash a : A$ there must be some other instance $\Gamma \vdash b : B$ such that $\Gamma \vdash a \ \sim_r \ b$.

$$\frac{\Gamma, x \overset{0}{:} A \vdash B \ \sim_r \ C}{\Gamma \vdash (x \overset{0}{:} A) \to B \ \sim_r \ C} \sim_r \texttt{FUNTYO} \qquad \frac{}{\Gamma \vdash \texttt{Nat} \ \sim_r \ \texttt{Nat}} \sim_r \texttt{NAT}$$

Fig. 3. Run-time equivalence rules on types

Relating constructors with erasure Figure 4 shows the rules for erased constructors. Erased functions are related to a version of their bodies without reference to the erased argument, the same procedure we used for function types. Erased vectors are related to equivalent list constructors.

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

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

Fig. 4. Constructor erasure rules

Relating arbitrary inductive constructors Note that we assume a correspondence between vectors and lists and thus a correspondence between their constructors. In a full system, correspondence of nominally distinct types would not be hardcoded, rather we would let the programmer state that two nominal types should be considered equivalent and confirm this via structural analysis. This analysis is performed by comparing the type signatures of aligned constructors.

The way we would do this for vectors is:

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:
   (a) Nil case: $\Gamma \vdash \texttt{Vec}\, A\, \texttt{z}^0 \sim_r \texttt{List}\ A$. Holds by assumption.
   (b) Cons case:

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

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

Thus, the algorithm for including new rules is: 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, introduce rules equating the nominal type formers and the constructors.

Relating eliminators with erasure Figure 5 shows that function applications get equated to the function term — which is equated to the body when the function is resolved to a lambda term. As for the data-type eliminators (see appendix B): If $\Gamma \vdash A \sim_r B$ then an eliminator for A is equated to an eliminator for B.

$$\frac{\Gamma \vdash f \sim_r f'}{\Gamma \vdash f \overset{0}{\cdot} a \sim_r f'} \sim_r \texttt{APP0}$$

Fig. 5. Erased application run-time equivalence rule

2.3 Simplifying QTT

We use a simplified version of quantitative type theory (QTT) [2] for our type theory. Our quantities are rather simple, only erased 0 and present ω. Typing judgements $\Gamma \vdash a \overset{\sigma}{:} A$ are indexed by a "mode" σ annotation that splits the type theory into two halves: one for compile time (erased) and one for run-time (present). These modes induce a certain flow or separation — erased values cannot be found in runtime position, but runtime values can be found in erased position. In order to enforce this, we make use of an ordering $0 \leq \omega$ as well as a multiplication operation $0\omega = 0$

2.4 Typing terms

Variables The only special condition that erasure induces in variables in our system is that erased variables may never end up in a run-time position[1] (because

[1] Note that run-time variables can be used in erased position, we only restrict flow in one direction.

they will not exist). The variable rule in Figure 6 enforces this invariant by requiring that the variable in the context have a usage greater-than or equal-to the type checking mode.

$$\frac{x \overset{\sigma'}{:} A \in \Gamma \quad \sigma \leq \sigma'}{\Gamma \vdash x \overset{\sigma}{:} A} \vdash \texttt{VAR}$$

Fig. 6. Typing rule for variables

Constructors Figure 7 shows the rules for constructors. The bodies of lambdas are checked in the extended context — where the added variable has the same usage annotation π as in the type annotation in the lambda-argument. Runid lambdas are well-typed if the usage-annotated version is well-typed according to the $\vdash$ `LAM` rule and if the function body is run-time-equivalent to its argument, $\Gamma, x \overset{\omega}{:} A \vdash b \sim_r x$.

$$\frac{\Gamma, x \overset{\pi}{:} A \vdash b \overset{\sigma}{:} B \quad \Gamma \vdash A \overset{0}{:} \texttt{Set}}{\Gamma \vdash \lambda(x \overset{\pi}{:} A).b \overset{\sigma}{:} (x \overset{\pi}{:} A) \to B} \vdash \texttt{LAM} \qquad \frac{\Gamma \vdash \lambda(x \overset{\omega}{:} A).b \overset{\sigma}{:} (x \overset{\omega}{:} A) \to B \quad \Gamma, x \overset{\omega}{:} A \vdash b \sim_r x}{\Gamma \vdash \lambda_r(x : A).b \overset{\sigma}{:} (x : A) \to_r B} \vdash \texttt{LAMR}$$

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

Fig. 7. Typing rules for constructors

Eliminators We give the rules for eliminators of natural numbers explicitly in Figure 8, the rules for other recursive data types can be derived in a similar way as shown in the appendix. The eliminator $\texttt{elNat}\, n\, P\, b\, c$ is typed in mode σ and takes four arguments:

1. The scrutinee n is type checked in the same position σ.
2. The motive P is typed in erased position, as it is a type function.
3. Branch b_z is the body for the zero case and is type checked in position σ.
4. Branch b_s is the body for the `suc` m case. We expose the value of m by extending the context.

For runid data eliminators, to reduce notational noise, we omit the typing hypothesis that the unmarked version of the eliminator is well-typed. Since eliminators imply something about the structure of the scrutinee in the relevant

branches, we only allow runid eliminators to be well-typed when the scrutinee is a variable x. We substitute into x the arbitrary constructor that matches the case — e.g. zero $b_z[x \mapsto \mathtt{z}]$ and successor $b_s[x \mapsto \mathtt{suc}\ m]$ branches.

For inductive data types (like `Nat`), we also substitute recursive subterms (like m in $\mathtt{suc}\ m$) with the result of the recursive eliminator call. The rule ensures the condition holds in the base cases (like `z`), and in the inductive cases (like $\mathtt{suc}\ m$) — where one assumes the subterms match the corresponding recursive results $\Gamma \vdash m \sim_r p$.

At first glance, we would prefer a structured solution to keep track of assumed run-time equivalences. In effect our inductive case is: $\Gamma \vdash p \sim_r m \implies \Gamma \vdash b_s \sim_r \mathtt{suc}\ m$. However, including such hypotheses would violate strict positivity in a formalization. Therefore we use substitution as a convenient definition for future formalization efforts. We suggest that future work give a more structured approach to defining, say, an assumption context Δ for a more general system.

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

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

Fig. 8. Typing rules for eliminators

When checking applications $f \overset{\pi}{\cdot} a$ in Figure 9 we multiply the mode σ by the annotation π when checking the argument to ensure that the usage of the function type aligns with the annotation. Similar to the $\vdash$ `LAMR` rule, runid applications are well-typed if the provided function is a well-typed runid function.

$$\frac{\Gamma \vdash f \overset{\sigma}{:} (x \overset{\pi}{:} A) \to B \quad \Gamma \vdash a \overset{\sigma\pi}{:} A}{\Gamma \vdash f \overset{\pi}{\cdot} a \overset{\sigma}{:} B} \vdash \mathtt{APP} \qquad \frac{\Gamma \vdash f \overset{\sigma}{:} (x : A) \to_r B \quad \Gamma \vdash a \overset{\sigma}{:} A}{\Gamma \vdash f \cdot_r a \overset{\sigma}{:} B} \vdash \mathtt{APPR}$$

Fig. 9. Typing rules for application

3 Denotational semantics of run-time equivalence

Our integration of runid into the type system hinges on our notion of two syntactic code fragments being "equal" at run-time — both in the sense of the result after erasure (structural) and the computational behaviour of runid fragments (extensional). We want to justify the validity of this syntactic judgement given that its meaning is essential to our analysis. To this end we give a denotational semantics, a semantic equality judgement $\Gamma \vDash a = b : A$ that tells us that two terms are equal at run-time.

To do this we establish a partial equivalence relation (PER) $\mathscr{a} \approx \mathscr{b} : \mathscr{A}$ in some semantic domain D which we interpret terms into — based on the model described by Abel in his habilitation thesis on normalization by evaluation (NbE) [1]. Unlike NbE we do not reify back into our original syntax, as we are only interested in equality in the domain.

3.1 Erasure

We define erasure as a partial function on the syntax of our language to an erasure-free subset of the same language. The function operates using the following algorithm recursively:

1. Terms marked erased are removed, e.g. $\downarrow(a \overset{0}{:} A) \to B = \downarrow B$
2. Terms marked runid have their marking removed, 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.

3.2 Domain values

Our domain is made up of two broad classes: D_V contains values that terms map to; D_T corresponds to type codes, i.e. values that represent types.

A value $d : D_V$ is either: a natural number $n : \mathbb{N}$, a function[2] $a \mapsto b : D \to D$[3], a pair $(a, b) : D \times D$ or an empty list symbol []. Accordingly, a type code $d_t : D_T$ is either: the $\mathtt{Nat}_D$ 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).

[2] We actually utilize Abels defunctionalization [1]: functions are closures and function application is a custom operation, we present these as regular functions for ease of reading.

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

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

Fig. 10. Values in D

3.3 Evaluating terms into domain values

We define an evaluation function $(\!|a|\!)_\gamma$ from terms into our domain D.

Evaluating variables The evaluation function maximally evaluates terms to simplify comparisons. This becomes a problem when dealing with open terms, as we have un-bound variables and are thus "stuck" in our evaluation. If we pretend we have a value for that variable we can reduce our term, so we simply quantify over the values assigned to variables. In practice this means providing our evaluation function with an environment γ that maps a given variable x to some value $\gamma(x)$.

Definition 1. *Variables are evaluated to the result of lookup in the environment*
$\gamma : \mathit{Env}$
$(\!|x|\!)_\gamma = \gamma(x)$

Evaluating constructors Lambdas are evaluated into semantic functions from a value a_v to some definition given by the evaluation of the lambda body. This body is a term that is weakened by the input variable a. To account for this we define an environment extension $\gamma[a \mapsto a_v]$ which binds the variable a to the provided argument value a_v.

Definition 2. *Constructors evaluate to values in D_V*

$$
\begin{aligned}
(\!|\lambda(a : A).b|\!)_\gamma &= a_v \mapsto (\!|b|\!)_{\gamma[a \mapsto a_v]} \\
(\!|(a, b)|\!)_\gamma &= ((\!|a|\!)_\gamma, (\!|b|\!)_{\gamma[x \mapsto (\!|a|\!)_\gamma]}) \\
(\!|\mathit{z}|\!)_\gamma &= 0 \\
(\!|\mathit{suc}\ n|\!)_\gamma &= 1 + (\!|n|\!)_\gamma \\
(\!|[]_l|\!)_\gamma &= [] \\
(\!|\mathit{cons}_l\, a\, b|\!)_\gamma &= (\!|h|\!)_\gamma,\ (\!|t|\!)_\gamma)
\end{aligned}
$$

Evaluating eliminators Since eliminators induce a computation procedure, when evaluating these we fully evaluate their result value. Lambda application evaluates to function application in the domain. 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 3. *Eliminators are evaluated to the result value of their computation*

$$
\begin{aligned}
(\!|f \cdot a|\!)_\gamma &= (\!|f|\!)_\gamma\,((\!|a|\!)_\gamma) \\
(\!|\mathtt{el} \times a\, P\, b|\!)_\gamma &= (\!|b|\!)_{\gamma[x \mapsto x_v, y \mapsto y_v]} \\
&\quad \textit{where: } (\!|a|\!)_\gamma = (x_v, y_v) \\
(\!|\mathtt{elNat}\, a\, \_\, b\, c|\!)_\gamma &= \mathtt{rec}_{\mathbb{N}}((\!|a|\!)_\gamma, (\!|b|\!)_\gamma, (k, r) \mapsto (\!|c|\!)_{\gamma[m \mapsto k, p \mapsto r]}) \\
&\quad \textit{where:} \\
&\quad \mathtt{rec}_{\mathbb{N}}(0, b, i) = b \\
&\quad \mathtt{rec}_{\mathbb{N}}(1 + m, b, i) = i(m, \mathtt{rec}_{\mathbb{N}}(m, b, i)) \\
(\!|\mathtt{elList}\, xs\, \_\, b\, c|\!)_\gamma &= \mathtt{rec}_L((\!|xs|\!)_\gamma, (\!|b|\!)_\gamma, (h, t, r) \mapsto (\!|c|\!)_{\gamma[a \mapsto h, as \mapsto t, p \mapsto r]}) \\
&\quad \textit{where:} \\
&\quad \mathtt{rec}_L([], b, i) = b \\
&\quad \mathtt{rec}_L((h, t), b, i) = i(h, t, \mathtt{rec}_L(t, b, i))
\end{aligned}
$$

Eliminators on recursive data are defined via recursion. For example natural numbers: the function is $rec_{\mathbb{N}}(a, b, i)$: a is the input number, 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. Note that these are simple domain functions and not a primitive recursion operations.

Evaluating types as terms Since we can encounter types as terms we need to map them to a domain value during evaluation. 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: 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 $F : D \to D_T$.

Definition 4. *Evaluation of types as terms*

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

3.4 Partial equivalence relations

Now that we have defined operations for creating domain values a_v, b_v we need some mechanism to relate values together in accordance with which type they correspond to $a_v \approx b_v : \mathcal{A}$. This is done by way of defining PERs $\mathcal{A} \subseteq D \times D$ on the domain — where each PER corresponds to a semantic type at run-time. PERs allow us to define an extensional notion of equality such that, for example, functions that behave the same but are not identical are still related to each other.

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, 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 $A \subset D$.

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.

Relating domain values We present the semantic types we have in an inductive fashion, building them "from the ground up" as Abel [1] does, courtesy of the assumed universe levels.

$\Pi(\mathscr{A}, \mathscr{F})$ denotes a semantic dependent function type where $\mathscr{A}$ is the argument type and $\mathscr{F} : \mathscr{A} \to \texttt{Per}$ the type family, such that $\mathscr{F}$ respects the relation $\mathscr{A}$ — It maps related values to the same relation: $\forall a \approx a' : \mathscr{A}.\mathscr{F}(a) = \mathscr{F}(a')$.

Definition 5 (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 6 (Semantic dependent product types: $\Sigma(\mathscr{A}, \mathscr{F})$). *Pairs are related pointwise, with dependency in the type specification.*

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

Definition 7 (Semantic types of inductive data). *Inductive constructors are related inductively. The base cases relate to themselves (e.g. 0) and the inductive cases relate if their sub-values relate. We write (h, n, t) as a shorthand for $(h, (n, t))$ for vector values.*

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

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

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

Relating type codes Different type codes may denote the same semantic type. Therefore we define the PERs on type codes (semantic universes) inductively, along with a corresponding lifting function [_] from type code to relation.

Definition 8 (Semantic type of type codes).

$$\frac{}{\mathtt{Nat}_D \approx \mathtt{Nat}_D : \mathcal{S}et}\ S-\mathtt{Nat}_D \qquad [\mathtt{Nat}_D] = \mathcal{N}at$$

$$\frac{A \approx A' : \mathcal{S}et \quad n \approx n' : \mathcal{N}at}{\mathtt{Vec}_D(A,n) \approx \mathtt{Vec}_D(A',n') : \mathcal{S}et}\ S-\mathtt{Vec}_D(,) \qquad [\mathtt{Vec}_D(A,n)] = \mathcal{V}ec([A],n)$$

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

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

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

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*

$$A \approx B : \mathcal{S}et$$

Then they lift to equal PER

$$[A] = [B]$$

Proof. We can prove this by induction on the rules. The base cases $S-\mathtt{Nat}_D$ and $\mathtt{Set}_D$ are trivial and the inductive cases hold by induction.

3.5 Defining a semantic run-time equality

We know have the necessary building blocks to give a semantic account of equivalence at run-time. This is done by way of three interpretation operations: one on terms, one on types and one on contexts.

Terms are interpreted into domain values, giving the semantic account of what a given term means.

Definition 9 (Interpretation of terms). *Terms are interpreted into their evaluation post-erasure*

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

Types are interpreted into a partial equivalence relation, giving us a semantic account of which values belong to a given semantic type and how we can compare two values of that type.

Definition 10 (Interpretation of types). *To interpret a syntactic type: erase, then evaluate to a type code, then lift to a semantic type.*

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

Contexts are intepreted into a PER on environments, letting us know that two environments contain semantically equivalent values.

Definition 11 (Interpretation of contexts). *Contexts are interpreted to the PER of related environments*

$$\frac{}{\emptyset \approx \emptyset : [\![\emptyset]\!]}\ S-NIL \qquad \frac{\gamma \approx \gamma' : [\![\Gamma]\!] \quad a \approx a' : [\![A]\!]_\gamma}{\gamma[x \mapsto a] \approx \gamma'[x \mapsto a'] : [\![\Gamma, x \overset{\omega}{:} A]\!]}\ S-EXTP$$

$$\frac{\gamma \approx \gamma' : \Gamma}{\gamma \approx \gamma' : \Gamma, x \overset{0}{:} A}\ S-EXT0$$

With these three operations we can define our account of semantic run-time equivalence.

Definition 12. *Terms are semantically run-time equivalent if for all related environments they interpret to equivalent values. Environments and values are determined equivalent by the PER obtained from interpreting the context and syntactic type respectively.*

$$\Gamma \vDash a = b : A \iff \forall \gamma \approx \gamma' : [\![\Gamma]\!].[\![a]\!]_\gamma \approx [\![b]\!]_{\gamma'} : [\![A]\!]_\gamma \tag{1}$$

4 Syntactically related terms are semantically related

We now show that our syntactic notion of run-time equivalence coincides with semantic equivalence in the PER model. Since the weak equivalence from the type system contains typing side-conditions and unrestricted reflexivity, it is not well-suited for an inductive argument. We therefore introduce a strong run-time relation that mirrors the structure of the semantic relation.

4.1 Strong run-time relation

The strong relation $\Gamma \vdash a \sim_s b : A$ consolidates typing, runid, and erasure side-conditions into a single judgement. Instead of a global reflexivity rule, it provides constructor-specific reflexivity and congruence rules, ensuring that only well-formed terms enter the relation. For erased and runid constructs, the rules equate each term with its optimized erased form or identity behavior, matching the semantic interpretation. This creates a syntax-directed relation that structurally aligns with the PERs. The exhaustive rules are listed in appendix D

Claim. If a is well typed and weakly related to b then it is strongly related to b.
$\Gamma \vdash a \overset{\omega}{:} A$ and $\Gamma \vdash a \ \sim_r \ b \implies \Gamma \vdash a \sim_s b : A$

4.2 Erasure does not fail

Because semantic interpretation begins by erasing terms, we must show that erasure is defined on all related terms. A straightforward induction on the strong relation establishes that no related term appears in an erased position. This removes the need to reason about partiality during the main proof.

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. $\downarrow a$ and $\downarrow b$ are defined*

Proof. We operate by induction on the strong rules:

1. Base cases:
   (a) variables x are only ever undefined 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.
2. Inductive case: We have the following inductive rules:
   (a) Congruence rules, e.g. $\Gamma \vdash \texttt{inl}^{\omega,\omega} a \sim_s \texttt{inr}^{\omega,\omega} b : A \uplus B$. These have a defined erasure if their subterm does, which holds by induction.
   (b) Erasure rules, e.g. $\Gamma \vdash \texttt{inl}^{\omega,0} a \sim_s b : A \,{}^{\omega}\!\uplus^{0} B$. The erasure function does not subrecurse on erased subterms, and the subterms marked not erased have a defined erasure by induction.
   (c) Runid rules. This also holds by induction, by analogous logic.
   (d) Erased constructors, i.e. erased left injections $\texttt{inl}^{0,\omega} a$ (and erased right injection). Neither of these are included in our relation.

   None of the inductive rules relate terms that would fail on erasure so $\downarrow a$ is defined.

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

4.3 Fundamental theorem

We prove that syntactically related terms are semantically related. The proof proceeds by induction on the derivation of the strong relation. PER rules (symmetry and transitivity) and constructor rules follow from induction. Eliminators interpret to the output of a function and thus follow from induction and extensionality. Validity of motives ensures that eliminators produce well-typed families in our semantics. Erased rules reduce to regular cases since erasure is total on related terms. Runid rules carry over because their semantic interpretation ignores the marker; the inductive step for runid eliminators (e.g. the eliminator on `Nat`) relies on lemmas equating our substitution with an induction principle. We give a more in depth proof in appendix E.

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 12 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 $[\![A]\!]_\gamma$

4.4 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). Thus fixing A to a specific type in each case.
2. Build monomorphization into the semantics, interpret the semantic type into an explicit quantification over valid semantic types $[\![(A \overset{0}{:} \texttt{Set}) \to (a \overset{\omega}{:} A) \to A]\!]_\gamma = \forall \mathscr{A}.\Pi(\mathscr{A}, \mathscr{A})$.
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 Related work

5.1 QTT

We elaborate on our simplification of QTT [2] here, specifically as it pertains to argument sharing. Argument sharing allows one to specify strict usage conditions

on variables; whether they be used zero times (i.e. used only at compile time), once (linearly), twice (etc), or an unrestricted number of times. This means that when checking subterms a, b in $a + b$ QTT will make sure that the linear variable x will only be used once in a single term by associating a custom context for each term and ensuring that in the sum of those contexts x is exactly linear, i.e. $\Gamma_1 \vdash a \overset{\omega}{:} \texttt{Nat}$, $\Gamma_2 \vdash b \overset{\omega}{:} \texttt{Nat}$ such that $x \overset{1}{:} \texttt{Nat} \in \Gamma_1 + \Gamma_2$.

This introduces some notational noise and downstream formalization headaches due to green slime — which limits pattern matching in proofs. Luckily we do not support anything beyond 0 and ω and can simplify the type theory. It suffices for us to know that $0 < \omega$ and have a multiplication operation $\sigma\pi$. $0 < \omega$ can be used to ensure that erased variables are never used in run-time position. Multiplication lets us "collapse" usages when any argument is erased as $0 = 0\sigma = \sigma 0$.

5.2 Ornaments

Ornaments [6] 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). Two existing approaches based on this are "Custom Representations" [9] and Agda2HS[4].

Custom representations Theocharis et al. [9] give a type theory for reasoning about the run-time representation of data generally; decoupling it from the theoretical structure of data. They generalize the practice of built-in binary representations of types e.g. natural numbers, allowing programmers to define a runtime representation of a type and giving a primitive conversion operation to transport along views.

These "refinements" carry with them proof obligations which must be partially provided by the programmer. 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.

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. This means we cannot equate structurally equivalent types, only having support for dependent pairs with erased index. 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.

5.3 Proof irrelevance

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

[5]. 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. For example, we can define a vector to have a `Prop` length index but cannot even write a safe `head` function — as it needs to eliminate the index which is forbidden for `Prop` values.

Ghost type theory [11] tries to alleviate these problems by way of a universe of "Ghost" types, which behave closer to run-time irrelevance, allowing for more granularity by discarding impossible branches, enabling the vector example we gave for Rocq. We recommend future work to explore the applicability of their results in our system. This was out of scope for this paper since they rely on proof-irrelevance rather than erasure.

5.4 Identity function detection in compiler backends

Some compilers already perform identity optimisations for commonplace shapes; Idris 2 recognizes identity functions on List-shaped things [8], 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.

The blog post [3] 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.

6 Conclusion and future work

We developed a core dependently typed language that separates run-time-relevant from erased components and marks run-time identity (runid) functions explicitly. A syntactic relation of run-time equivalence was justified by a PER-based NbE semantics, and an erasure translation made precise which parts of a program survive into execution. We proved that syntactically related erased terms are semantically equal after erasure. Several extensions remain open: supporting inductive families more generally, giving a principled account of assumptions and higher-order runid functions beyond our substitution-based approach, and mechanizing the development to validate the boundaries of the system. Together, these results provide a foundation for reasoning about modalities on run-time behaviour in dependent type theory and for optimizing dependently typed programs.

Acknowledgments. This paper is a modified version of the Master thesis presented by the principal author[4].

[4] https://resolver.tudelft.nl/uuid:e6d5e4a8-6df5-4867-ad9b-4ac77cdc2512

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Abel, A.: Normalization by evaluation: Dependent types and impredicativity. Habilitation. Ludwig-Maximilians-Universität München (2013)
2. Atkey, R.: Syntax and semantics of quantitative type theory. In: Dawar, A., Grädel, E. (eds.) Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018. pp. 56–65. ACM (2018). https://doi.org/10.1145/3209108.3209189, http://doi.acm.org/10.1145/3209108.3209189
3. Boitel, L.: Detecting identity functions in flambda. https://ocamlpro.com/blog/2021_07_16_detecting_identity_functions_in_flambda/, [Accessed 24-02-2025]
4. Cockx, J., Melkonian, O., Escot, L., Chapman, J., Norell, U.: Reasonable agda is correct haskell: writing verified haskell using agda2hs. In: Polikarpova, N. (ed.) Haskell '22: 15th ACM SIGPLAN International Haskell Symposium, Ljubljana, Slovenia, September 15 - 16, 2022. pp. 108–122. ACM (2022). https://doi.org/10.1145/3546189.3549920, https://doi.org/10.1145/3546189.3549920
5. Letouzey, P.: A new extraction for coq. In: Geuvers, H., Wiedijk, F. (eds.) Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers. Lecture Notes in Computer Science, vol. 2646, pp. 200–219. Springer (2002), http://link.springer.de/link/service/series/0558/bibs/2646/26460200.htm
6. McBride, C.: Ornamental algebras, algebraic ornaments. Journal of functional programming **47** (2010)
7. McBride, C., McKinna, J.: The view from the left. Journal of Functional Programming **14**(1), 69–111 (2004). https://doi.org/10.1017/S0956796803004829, http://dx.doi.org/10.1017/S0956796803004829
8. Stafford, Z.: 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 (2025), [Accessed 13-08-2025]
9. Theocharis, C., Brady, E.: Custom representations of inductive. In: Trends in Functional Programming: 26th International Symposium, TFP 2025, Oxford, UK, January 14–16, 2025, Revised Selected Papers. p. 302. Springer Nature (2025)
10. Wadler, P.: Views: A way for pattern matching to cohabit with data abstraction. In: POPL. pp. 307–313 (1987)
11. Winterhalter, T.: Dependent ghosts have a reflection for free. Proceedings of the ACM on Programming Languages **8**(ICFP), 630–658 (2024). https://doi.org/10.1145/3674647, https://doi.org/10.1145/3674647

A Language

A.1 Syntax

Type formers track usage information in two ways: explicit annotations which bind a variable, e.g. $(x \overset{\sigma}{:} A) \to B$ for functions, and position-based which track the usage of terms in types, e.g. $\texttt{Vec}\, A\, n^{\sigma}$. Position-based annotations are a result

of our hardcoded type examples, rather than fully general inductive families, e.g. $\texttt{Vec} \ : (A \overset{0}{:} \texttt{Set}) \to (n \overset{\sigma}{:} \texttt{Nat}) \to \texttt{Set}$ where indices are always explicitly bound.

Constructors and eliminators of a given type have analogous annotations vis-a-vis usages, e.g. sums $A \, {}^{\pi}\uplus^{\rho} B$ with constructors $\texttt{inl}^{\pi,\rho} a$, $\texttt{inr}^{\pi,\rho} b$ and eliminator $\texttt{el} \, {}^{\pi}\uplus^{\rho} \, s \, P \, c \, d$.

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

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

$A, B, P ::= (x \overset{\sigma}{:} A) \to B \mid (x : A) \to_r B$	Dependent functions
$\mid \quad \texttt{List} \ A \quad \mid \quad \texttt{Vec} \, A \, n^{\sigma}$	Recursive types
$\mid \quad A \, {}^{\sigma}\uplus^{\rho} B \quad \mid \quad (x \overset{\sigma}{:} A) \times B^{\rho}$	Sum and product type
$\mid \quad \texttt{Nat}$	Natural Numbers
$\mid \quad \texttt{Set}$	Universes
$a, b, c, d, n, P ::= x$	Variable
$\mid \lambda(x \overset{\pi}{:} A).b \mid a \overset{\pi}{\cdot} c \mid \lambda_r(x : A).b \mid a \cdot_r b$	Functions
$\mid {}^{\pi}(a, b)^{\rho} \mid \texttt{el} \, {}^{\pi}\times^{\rho} \, a \, P \, b \mid \texttt{el}_r \, {}^{\pi}\times^{\rho} \, a \, P \, b$	Products
$\mid \texttt{suc} \ a \mid \texttt{z} \mid \texttt{elNat} \, a \, P \, b \, c \mid \texttt{el}_r\texttt{Nat} \, a \, P \, b \, c$	Natural Numbers
$\mid \texttt{cons}_l \, a \, b \mid \texttt{[]}_l \mid \texttt{elList} \, a \, A \, P \, b \, c \mid \texttt{el}_r\texttt{List} \, a \, A \, P \, b \, c$	Lists
$\mid \texttt{cons}_v^{\pi} \, a \, n \, b \mid \texttt{[]}_v^{\pi} \mid \texttt{elVec}^{\pi} \, a \, P \, c \, d \mid \texttt{el}_r\texttt{Vec}^{\pi} \, a \, P \, c \, d$	Vecs
$\mid \texttt{inl}^{\pi,\rho} a \mid \texttt{inr}^{\pi,\rho} b \mid \texttt{el} \, {}^{\pi}\uplus^{\rho} \, a \, P \, c \, d \mid \texttt{el}_r \, {}^{\pi}\uplus^{\rho} \, a \, P \, c \, d$	Sums

Fig. 11. Syntax

A.2 Constructor typing rules

Constructors with usage annotations multiply the mode σ by the usage when checking annotated terms — e.g. when checking the left value a in ${}^{\pi}(a, b)^{\rho}$ we multiply the checking by π, i.e. $\sigma\pi$. 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. Constructors that do not carry usage annotations maintain the mode when checking their arguments.

In order to define the type for dependent pairs we must substitute the concrete left value into the right type. Substitution is defined as $a[x \mapsto b]$ where each instance of variable x in b is substituted for a.

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

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

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

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

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

Fig. 12. Typing rules for constructors

A.3 Eliminator typing rules

This multiplication 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 multiplied to match the usage of the constructor: if the left constructor is erased then its corresponding branch must also be in erased position.

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

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

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

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

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

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

Fig. 13. Typing rules for eliminators

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

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

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

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

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

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

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

Fig. 14. Typing rules for runid eliminators

A.4 Forming types

Type formers are always typed in erased position $\Gamma \vdash A \overset{0}{:} \texttt{Set}$, as type formation is only relevant to the compile time. Dependent pairs differ from those in [2] in that we admit usage annotation for both types, while Atkey's dependent tensors only admit usages for the first argument. The rules are standard

$$\frac{\Gamma \vdash A \overset{0}{:} \texttt{Set} \quad \Gamma, x \overset{\pi}{:} A \vdash B \overset{0}{:} \texttt{Set}}{\Gamma \vdash (x \overset{\pi}{:} A) \to B \overset{0}{:} \texttt{Set}} \vdash \Pi \qquad \frac{\Gamma \vdash A \overset{0}{:} \texttt{Set} \quad \Gamma, x \overset{\omega}{:} A \vdash B \overset{0}{:} \texttt{Set}}{\Gamma \vdash (x : A) \to_r B \overset{0}{:} \texttt{Set}} \vdash \Pi_r$$

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

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

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

Fig. 15. Type rules for type formers

A.5 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"[2]. 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.

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

Fig. 16. Typing rule to shift type checking mode.

B Runtime equivalence relation

B.1 Types

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

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

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

Fig. 17. Run-time equivalence rules on types

B.2 Constructors

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.

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

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

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

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

Fig. 18. Constructor erasure rules

B.3 Eliminators

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.

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

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

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

Fig. 19. Eliminator erasure rules

C Erasure function

(Types)

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

(Constructors)

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

(Eliminators)

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

D Strong runtime equivalence relation

D.1 Types

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

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

Fig. 20. Strong relation on function types

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

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

Fig. 21. Strong relation on product types

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

Fig. 22. Strong relation on list type

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

Fig. 23. Strong relation on vector type

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

Fig. 24. Strong relation on nat type

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

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

Fig. 25. Strong relation on sum types

D.2 Constructors

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

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

Fig. 26. Strong relation on lambdas

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

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

Fig. 27. Strong relation on pairs

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

Fig. 28. Strong relation on `Nat` constructors

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

Fig. 29. Strong relation on `List` constructors

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

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

Fig. 30. Strong relation on `Vec` constructors

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

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

Fig. 31. Strong relation on sum injections

D.3 Eliminators

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

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

Fig. 32. Strong relation for function application

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

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

Fig. 33. Strong relation for `Nat` eliminator

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

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

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

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

Fig. 34. Strong relation for $\times$ eliminator

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

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

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

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

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

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

Fig. 35. Strong relation for $\uplus$ eliminators

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

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

Fig. 36. Strong relation for `List` eliminators

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

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

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

Fig. 37. Strong relation on vector eliminators

E Fundamental theorem

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

Theorem 3. *Syntactic run-time equivalence implies semantic run-time equivalence*

$\Gamma \vdash a \sim_s b : A \implies \Gamma \vDash a = b : A$

By Definition 12 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 E.1, then go over the proof strategy for regular rules subsection E.2, rules for erasure subsection E.3 and finally rules for runid terms subsection E.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.

E.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.

E.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 8 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. $\Gamma \vdash \mathtt{z} \sim_s \mathtt{z} : \mathtt{Nat}$ 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 $\mathtt{suc}\ \ a$ 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).

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

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

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

By evaluation we need to prove

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

By congruence on semantic Nats (**??**) it suffices to prove

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

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 5) 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 5, 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

$$P \approx P' : \Pi(\mathscr{A}, \mathscr{Set})$$

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 5 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

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

$$b \approx b' : \mathcal{P}(z)$$

$$c \approx c' : \Pi(\mathcal{Nat}, n \mapsto \Pi(\mathcal{P}(n), \_ \mapsto \mathcal{P}(1+n)))$$

It holds that

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

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

1. Base case: $0 \approx 0 : \mathcal{Nat}$.
   We need to prove that

   $$\mathtt{rec}_{\mathbb{N}}(0,b,c) \approx \mathtt{rec}_{\mathbb{N}}(0,b',c') : \mathcal{P}(0)$$

   Which computes to

   $$b \approx b' : \mathcal{P}(0)$$

   Which holds by assumption.
2. Inductive case: $1+k \approx 1+k' : \mathcal{Nat}$.
   Given the induction hypothesis:

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

   We need to prove that

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

   If we compute our proof term we get

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

   By assumption it holds that

   $$c \approx c' : \Pi(\mathcal{Nat}, n \mapsto \Pi(\mathcal{P}(n), \_ \mapsto \mathcal{P}(1+n)))$$

   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).

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

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
$$[\![a]\!]_\gamma \approx [\![a']\!]_{\gamma'} : \mathscr{Nat}$$
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 5). For arbitrary $k \approx k' : \mathscr{Nat}$, $r \approx r' : \mathscr{P}(k)$ the functions are related.
$$[\![c]\!]_{\gamma[m\mapsto k, p\mapsto r]} \approx [\![c']\!]_{\gamma[m\mapsto k', p\mapsto r']} : \mathscr{P}(1+k)$$

E.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

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

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

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

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

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

Proof. We want to prove, after erasing, that

$$([\![h]\!]_\gamma, [\![t]\!]_\gamma) \approx ([\![h']\!]_\gamma, [\![t']\!]_\gamma) : \mathcal{L}ist(\mathcal{A}) \tag{2}$$

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. $[\![t]\!]_\gamma \approx [\![t']\!]_{\gamma'} : \mathcal{L}ist(\mathcal{A})$ 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

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

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 5) 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.

E.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 5): 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

$$i(k, \mathtt{rec}_{\mathbb{N}}(k, b, i)) \approx 1 + k' : \mathcal{Nat} \tag{3}$$

Using the induction hypothesis and the third assumption we get

$$i(k, k') \approx 1 + k' : \mathcal{Nat} \tag{4}$$

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

$$i(k, \mathtt{rec}_{\mathbb{N}}(k, b, i)) \approx i(k, k') : \mathcal{Nat} \tag{5}$$

By transitivity, combining Equation 4 and Equation 5 gives us the exact proof statement we need in Equation 3

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

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

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

$$\texttt{rec}_{\mathbb{N}}(k, [\![b]\!]_{\gamma[a \mapsto k]}, (n, r) \mapsto [\![c]\!]_{\gamma[a \mapsto k, m \mapsto n, p \mapsto r]}) \approx k' : \mathcal{N}at \tag{6}$$

   We must prove that

$$[\![c]\!]_{\gamma[[a \mapsto k, m \mapsto k, p \mapsto k]} \approx 1 + k' : \mathcal{N}at \tag{7}$$

   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