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. Dependent types track precise properties of programs at compile-time, but they can also impose a run-time overhead. Erasure annotations from Quantitative Type Theory (QTT) reduce this overhead by distinguishing compile-time from run-time data. Since different types in QTT can share the same run-time representation, functions converting between such types behave as identity functions at run-time but are not recognized as such, so a run-time overhead compared to the equivalent simply-typed program remains.
We introduce a type theory extending a fragment of QTT with a principled notion of run-time identity (runid) functions. Our language is equipped with a syntactic notion of run-time equivalence and a denotational semantics describing the run-time behaviour of programs. We argue that these annotations correctly capture when terms are equivalent at run time, establishing that runid functions can safely be replaced by the identity function at run-time.

Keywords: Dependent types · Inductive families · Structural typing.

1 Introduction

The presence of bugs is a constant reality of software development, and society often pays a high cost to mitigate their consequences. Formal verification allows us to reduce this cost by proving that software adheres to its specification. However, formal verification comes with its own costs — often including worse run-time performance of the final program in particular.

Dependent type systems are one established way to formally verify software. To manage the overhead of proofs at run-time, they can include *erasure annotations* [2], which allow programmers to communicate to the compiler that certain parts can be safely removed during compilation. For example, Listing 1 shows how to erase the length index of vectors in Agda [11] by annotating it with `@0`.

To capture different invariants of data at different points in the program, programmers often have to define multiple data types that have the same run-time representation [13,8]. While the programmer might know that two types share a run-time representation, it is not possible to reflect this knowledge back into the type system. For example, the `listToVec` function defined in Listing 2

```
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: Definition of vector with erased length index in Agda.

returns its input unchanged at run-time, but it does so by traversing the whole list. This additional overhead is what we aim to eliminate in this paper.

```
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.

Theocharis and Brady [12] provide a way to work around this problem by relying on a custom representation of data types — with some limitations. To use this approach, programmers have to manually write mappings from every data type into its basic unornamented version, and manually prove certain conditions that are implicit in the data type. For example, vectors would be encoded as $\texttt{Vec}\,A\,n = \{l : \texttt{List}\,A \mid \texttt{length}(l) = n\}$, and the programmer has to manually prove that the length of a vector aligns with the index. We instead aim for a "plug and play" solution that works on existing data types with minimal annotations.

For a practical idea of what we aim for, consider the example in Listing 3 which uses a possible future extension of Agda with two new annotations `@repr=...` and `@runid`. 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 to a `List` and that `listToVec` should be compiled to an identity function. The compiler can verify the validity of these annotations automatically by inspecting the code. It then uses this newly confirmed information to optimize the code that uses these definitions. We treat runid functions as first-class citizens of the type system so we can also define functions whose runid status depends on the runid status of one of its inputs, such as the `mapId` function defined in Listing 4. These annotations provide the type system with sufficient information to perform a structural analysis which verifies run-time equivalences and thus validity of runid functions. In particular, there is no need to manually define types as refinements of the base type, nor to provide manual coherence proofs.

```
data @repr=List Vec (A : Set) : @0 ℕ → Set where
    @repr=[]
    []  : Vec A zero
    @repr=_::_
    _::_ : ∀ {@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 example of a runid function.

```
@runid mapId : (@runid f : A -> B) -> List A -> List B
mapId f [] = []
mapId f (a :: as) = f a :: mapId f as
```

Listing 4: Mapping a runid function over a list.

Contributions. In this paper we focus on defining a core type theory and leave elaboration to future work. As such our contributions are:

- We define a core type theory with annotations for erasure and a type of run-time identity (runid) functions, which relies on a syntactic run-time equivalence relation $\Gamma \sim_r \Gamma' \mathbin{;} \Delta \vdash a \sim_r b : A \sim_r B$ for its typing rules (subsection 2.2).
- We provide a denotational semantics of our type theory and the syntactic run-time equivalence relation using an extensional semantic domain based on a PER model [1] (section 3).
- We give a proof sketch of the main soundness property of our language: terms related by our run-time equivalence relation are mapped to equal values in our semantic domain, justifying the compilation of runid functions to the identity function (section 4).

2 Extending QTT with Runid Functions

To define our type theory, we take as a starting point Quantitative Type Theory (QTT) [2] with two quantities 0 (for erased arguments) and ω (for non-erased arguments) as well as Agda-style universe levels. As primitive types, we take the type `Nat` of natural numbers, the type `List` A of lists, and the type `Vec` $A\,n$ of vectors with erased length index. We extend this type theory with a type of run-time identity functions $(x : A) \to_r B$. To define the typing rules for them, we first define a run-time equivalence relation on terms. We rely on the Barendregt convention in this paper, thus ignoring variable name capture.

2.1 Syntax

Figure 1 defines the syntax of our calculus. Functions carry a usage annotation π for their argument x in both the type $(x \overset{\pi}{:} A) \to B$, introduction $\lambda(x \overset{\pi}{:} A).\, b$ and elimination $f \overset{\pi}{\cdot} a$. Runid functions cannot take erased input, so they do not require a usage annotation — neither in the type $(x : A) \to_r B$, introduction $\lambda_r(x : A).\, b$, nor in the elimination $f \cdot_r a$. Natural numbers $\texttt{Nat}$ are Peano natural numbers with constructors for zero $\texttt{z}$ and successor $\texttt{suc}$. The elimination principle is written as $\texttt{elNat}\, a\, P\, b\, (m\, p.\, c)$ where a is the scrutinee, P the motive, b the base case and c the inductive case — exposing variables m and p. It also has a runid variant $\texttt{elNat}_r\, a\, P\, b\, (m\, p.\, c)$, which will be compiled to its scrutinee a. Vectors and lists are defined analogously. Contexts $\Gamma = \Gamma', x \overset{\sigma}{:} A$ map each variable x to both its type A and usage σ.

$i, j, k \in \mathbb{N}$	Levels
x, y, z, p	Variables
$\pi, \sigma, \rho ::= 0 \mid \omega$	Quantities
$\Gamma ::= \emptyset \mid \Gamma, x \overset{\pi}{:} A$	Contexts
$A, B ::= (x \overset{\pi}{:} A) \to B \mid (x : A) \to_r B \mid \texttt{List}\ A \mid \texttt{Vec}\, A\, n \mid \texttt{Nat} \mid \texttt{Set}_i$	Types
$a, b, c, P ::= x$	Variables
$\mid \lambda(x \overset{\pi}{:} A).\, b \mid \lambda_r(x : A).\, b \mid a \overset{\pi}{\cdot} c \mid a \cdot_r b$	Functions
$\mid \texttt{suc}\, a \mid \texttt{z} \mid \texttt{elNat}\, a\, P\, b\, (m\, p.\, c) \mid \texttt{elNat}_r\, a\, P\, b\, (m\, p.\, c)$	$\texttt{Nat}$
$\mid \texttt{cons}_l\, a\, b \mid \texttt{[]}_l \mid \texttt{elList}\, a\, P\, b\, (h\, t\, p.\, c) \mid \texttt{elList}_r\, a\, P\, b\, (h\, t\, p.\, c)$	$\texttt{List}$
$\mid \texttt{cons}_v\, a\, n\, b \mid \texttt{[]}_v \mid \texttt{elVec}\, a\, P\, c\, (h\, n\, t\, p.\, d) \mid \texttt{elVec}_r\, a\, P\, c\, (h\, n\, t\, p.\, d)$	$\texttt{Vec}$

Fig. 1. Syntax

2.2 Run-time Equivalence

This section covers our first main contribution: the run-time equivalence relation $\Gamma \sim_r \Gamma' \,;\, \Delta \vdash a \sim_r b : A \sim_r B$, stating that two well typed terms $\Gamma \vdash a \overset{\omega}{:} A$ and $\Gamma' \vdash b \overset{\omega}{:} B$ have the same value or behaviour at run-time, which we make use of in our typing rules for runid terms. The assumption context Δ tells us which variables are to be considered run-time equivalent. Δ always relates at least each non-erased variable from Γ with its counterpart from Γ'. For example, if $\Gamma = x \overset{0}{:} A, y \overset{\omega}{:} B$ and $\Gamma' = z \overset{\omega}{:} C$, then $\Delta = y \sim_r z$. The typing rules for the runid eliminators of the primitive types also introduce additional assumptions to the

context in their typing rules. To improve readability, we omit the (assumption) contexts but only include the (assumption) context *extensions*.

$$\frac{x \overset{\omega}{:} A \in \Gamma \qquad y \overset{\omega}{:} B \in \Gamma' \qquad x \sim_r y \in \Delta}{\Gamma \sim_r \Gamma' \,;\, \Delta \vdash x \sim_r y : A \sim_r B}$$

Fig. 2. Assumption context rule

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. Figure 3 applies this to functions and eliminators: runid lambdas are related to the identity function and runid eliminators are related to their argument.

$$\frac{}{\vdash (x : A) \to_r B \sim_r (x : A) \to_r A : \mathtt{Set}_i \sim_r \mathtt{Set}_j}$$

$$\frac{}{\vdash \lambda_r(x : A).\, b \sim_r \lambda_r(x : A).\, x : (x : A) \to_r B \sim_r (x : A) \to_r A}$$

$$\frac{\vdash a \sim_r a' : A \sim_r A}{\vdash f \cdot_r a \sim_r a' : B \sim_r A} \qquad \frac{\vdash x \sim_r x' : \mathtt{Nat} \sim_r \mathtt{Nat}}{\vdash \mathtt{elNat}_r\, x\, P\, b\, (m\, p.\, c) \sim_r x' : P \overset{\omega}{\cdot} x \sim_r \mathtt{Nat}}$$

$$\frac{\vdash x \sim_r x' : \mathtt{List}\ A \sim_r \mathtt{List}\ A}{\vdash \mathtt{elList}_r\, x\, P\, b\, (h\, t\, p.\, c) \sim_r x' : P \overset{\omega}{\cdot} x \sim_r \mathtt{List}\ A}$$

Fig. 3. Relating runid terms

Relating Types with Erasure. Figure 4 shows the rules for types. Dependent function types weaken the codomain of the result type with the argument variable $\Gamma, x \overset{0}{:} A \vdash B \overset{0}{:} \mathtt{Set}_i$. We cannot just write $\vdash (x \overset{0}{:} A) \to B \sim_r B : \mathtt{Set}_i \sim_r \mathtt{Set}_i$ since B might make use of the variable x. Instead we pick a type C that does not refer to x but is run-time equivalent to $B(x)$.

$$\frac{x \overset{0}{:} A \vdash B \sim_r C \ : \ \mathtt{Set}_i \sim_r \mathtt{Set}_i}{\vdash (x \overset{0}{:} A) \to B \sim_r C \ : \ \mathtt{Set}_i \sim_r \mathtt{Set}_i} \qquad \frac{}{\vdash \mathtt{Nat} \sim_r \mathtt{Nat} \ : \ \mathtt{Set}_0 \sim_r \mathtt{Set}_0}$$

$$\frac{\vdash A \sim_r A' \ : \ \mathtt{Set}_i \sim_r \mathtt{Set}_i}{\vdash \mathtt{Vec}\, A\, n \sim_r \mathtt{List}\ A' \ : \ \mathtt{Set}_i \sim_r \mathtt{Set}_i}$$

Fig. 4. Run-time equivalence rules on types

Relating Constructors with Erasure. Figure 5 shows the rules for erased constructors. Erased functions are related to a version of their bodies without reference to the erased argument, following the same procedure we used for function types. Vectors with an erased index are related to equivalent list constructors.

$$\frac{x \overset{0}{:} A \vdash b \sim_r c \ : \ B \sim_r C}{\vdash \lambda(x \overset{0}{:} A).\, b \sim_r c \ : \ (x \overset{0}{:} A) \to B \sim_r C} \qquad \frac{}{\vdash []_v \sim_r []_l \ : \ \mathtt{Vec}\, A\, \mathtt{z} \sim_r \mathtt{List}\ A}$$

$$\frac{\vdash a \sim_r b \ : \ A \sim_r A \qquad \vdash as \sim_r bs \ : \ \mathtt{Vec}\, A\, n \sim_r \mathtt{List}\ A}{\vdash \mathtt{cons}_v\, a\, n\, as \sim_r \mathtt{cons}_l\, b\, bs \ : \ \mathtt{Vec}\, A\, (\mathtt{suc}\, n) \sim_r \mathtt{List}\ A}$$

Fig. 5. Constructor erasure rules

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, first comparing the signatures of the type formers and then comparing the signatures of the constructors, assuming equivalence of the type formers.

Relating Eliminators with Erasure. Figure 6 shows that applications of erased functions get related to the function itself, erasing the argument. As for the eliminators: the eliminator for vectors is related to the eliminator for lists.

Relating Terms over Conversions (Figure 7). Since we wish to be able to relate any well typed term (at least to itself), we add a conversion rule to stipulate that we can relate terms at one type if they are related at a convertible type. Our conversion rule is standard, relying on an untyped conversion relation $A \equiv B$. The definition of conversion for our type theory is the standard notion of

$$\frac{\vdash f \overset{0}{\cdot} a \sim_r c : B[x \mapsto a] \sim_r C}{\vdash f \sim_r c : (x \overset{0}{:} A) \to B \sim_r C}$$

$$\frac{\begin{array}{c}\vdash a \sim_r a' : \mathtt{Vec}\, A\, k \sim_r \mathtt{List}\ A' \\ \vdash P \sim_r P' : (n \overset{0}{:} \mathtt{Nat}) \to (x \overset{\omega}{:} \mathtt{Vec}\, A\, n) \to \mathtt{Set}_i \sim_r (x' \overset{\omega}{:} \mathtt{List}\ A') \to \mathtt{Set}_i \\ \vdash b_n \sim_r b_n' : P \overset{0}{\cdot} \mathtt{z} \overset{\omega}{\cdot} []_v \sim_r P' \overset{\omega}{\cdot} []_l \\ h : A, n \overset{0}{:} \mathtt{Nat}, t : \mathtt{Vec}\, A\, n, p : P \overset{0}{\cdot} n \overset{\omega}{\cdot} t \sim_r h' : A, t' : \mathtt{List}\ A', p' : P' \overset{\omega}{\cdot} t' \\ \vdash b_c \sim_r b_c' : P \overset{0}{\cdot} (\mathtt{suc}\, n) \overset{\omega}{\cdot} \mathtt{cons}_v\, h\, n\, t \sim_r P' \overset{\omega}{\cdot} \mathtt{cons}_l\, h'\, t'\end{array}}{\vdash \mathtt{elVec}\, a\, P\, b_n\, (h\, n\, t\, p.\, b_c) \sim_r \mathtt{elList}\, a'\, P'\, b_n'\, (h'\, t'\, p'.\, b_c') : P \overset{0}{\cdot} k \overset{\omega}{\cdot} a \sim_r P' \overset{\omega}{\cdot} a'}$$

Fig. 6. Erased application run-time equivalence rule

conversion including beta-reduction and congruence rules and is omitted here. In particular, we do *not* assert general convertibility of run-time equivalent terms.

$$\frac{\vdash a \sim_r b : A \sim_r B \qquad A \equiv A' \qquad B \equiv B'}{\vdash a \sim_r b : A' \sim_r B'}$$

Fig. 7. Conversion rule for run-time equivalence

2.3 Typing Terms

Typing judgements $\Gamma \vdash a \overset{\sigma}{:} A$ are indexed by a mode annotation σ that splits the type theory into two halves: one for compile time (erased) and one for run-time (present) [2]. These modes enforce that information only flows one way: erased values cannot be found in run-time position, but run-time values can be found in erased position. To enforce this, the typing rules make use of the ordering on quantities where $0 \leq \omega$, as well as a multiplication operation defined by $0\omega = 0 = \omega 0$ and $\omega\omega = \omega$.

Variables (Figure 8). 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. The variable rule enforces this invariant by requiring that the variable in the context have a usage greater-than or equal-to the type checking mode.

Types (Figure 9). Types are well typed terms of type $\mathtt{Set}_i$ if all their components are well typed. In the rule for function types, the variable x is bound with

$$\frac{x \overset{\sigma'}{:} A \in \Gamma \qquad \sigma \leq \sigma'}{\Gamma \vdash x \overset{\sigma}{:} A} \qquad\qquad \frac{\Gamma \vdash a \overset{\sigma}{:} A \qquad A \equiv B}{\Gamma \vdash a \overset{\sigma}{:} B}$$

Fig. 8. Typing rule for variables and conversion rule for typing

the quantity π in the codomain. In particular, if the function type is erased then it can only appear in erased positions in the codomain.

$$\frac{}{\vdash \mathtt{Nat} \overset{\sigma}{:} \mathtt{Set}_0} \qquad \frac{\vdash A \overset{\sigma}{:} \mathtt{Set}_i}{\vdash \mathtt{List}\ A \overset{\sigma}{:} \mathtt{Set}_i} \qquad \frac{\vdash A \overset{\sigma}{:} \mathtt{Set}_i \qquad \vdash n \overset{0}{:} \mathtt{Nat}}{\vdash \mathtt{Vec}\ A\ n \overset{\sigma}{:} \mathtt{Set}_i}$$

$$\frac{\vdash A \overset{\sigma}{:} \mathtt{Set}_i \qquad x \overset{\pi}{:} A \vdash B \overset{\sigma}{:} \mathtt{Set}_j}{\vdash (x \overset{\pi}{:} A) \to B \overset{\sigma}{:} \mathtt{Set}_{\max(i,j)}} \qquad \frac{\vdash A \overset{\sigma}{:} \mathtt{Set}_i \qquad x \overset{\omega}{:} A \vdash B \overset{\sigma}{:} \mathtt{Set}_j}{\vdash (x : A) \to_r B \overset{\sigma}{:} \mathtt{Set}_{\max(i,j)}}$$

$$\frac{}{\vdash \mathtt{Set}_i \overset{\sigma}{:} \mathtt{Set}_{(i+1)}}$$

Fig. 9. Typing rules for type formers

Constructors (Figure 10). 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 (as for normal lambdas) and the function body is run-time equivalent to its argument.

We diverge from the standard formulation of QTT by requiring that the type annotation A is available at quantity σ instead of 0. This means it is impossible to assign a type to functions with erased type parameters that are used in the codomain, e.g. we cannot implement a function of type $(A \overset{0}{:} \mathtt{Set}_i) \to (a \overset{\omega}{:} A) \to A$. The reason is that the type annotation is still present in the result of erasure (see Figure 14). Even if we allowed such functions, they could still not be related to any function without an erased type parameter input, since then it is required to annotate the lambda abstraction. We see three possible approaches to lifting this restriction:

1. We could relate functions with erased type parameters to a monomorphized type, e.g. by relating $(A \overset{0}{:} \mathtt{Set}_i) \to (a \overset{\omega}{:} A) \to A$ to $(a \overset{\omega}{:} \mathtt{Nat}) \to \mathtt{Nat}$. This would require careful handling to avoid relating distinct concrete instantiations of a polymorphic type.

$$\frac{x \overset{\pi}{:} A \vdash b \overset{\sigma}{:} B \qquad \vdash A \overset{\sigma}{:} \mathtt{Set}_i}{\vdash \lambda(x \overset{\pi}{:} A).\, b \overset{\sigma}{:} (x \overset{\pi}{:} A) \to B}$$

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

$$\frac{}{\vdash \mathtt{z} \overset{\sigma}{:} \mathtt{Nat}} \qquad \frac{\vdash n \overset{\sigma}{:} \mathtt{Nat}}{\vdash \mathtt{suc}\, n \overset{\sigma}{:} \mathtt{Nat}} \qquad \frac{}{\vdash []_l \overset{\sigma}{:} \mathtt{List}\ A} \qquad \frac{\vdash h \overset{\sigma}{:} A \qquad \vdash t \overset{\sigma}{:} \mathtt{List}\ A}{\vdash \mathtt{cons}_l\, h\, t \overset{\sigma}{:} \mathtt{List}\ A}$$

$$\frac{}{\vdash []_v \overset{\sigma}{:} \mathtt{Vec}\, A\, \mathtt{z}} \qquad \frac{\vdash h \overset{\sigma}{:} A \qquad \vdash t \overset{\sigma}{:} \mathtt{Vec}\, A\, n \qquad \vdash n \overset{0}{:} \mathtt{Nat}}{\vdash \mathtt{cons}_v\, h\, n\, t \overset{\sigma}{:} \mathtt{Vec}\, A\, (\mathtt{suc}\, n)}$$

Fig. 10. Typing rules for constructors

2. We could instead extend our type theory with the 'any' type from gradual type theory [9], and use that type to instantiate erased type parameters.
3. We could instead directly target an untyped language with our erasure.

We leave these investigations for future work, and for now assume type parameters are erased in a separate compiler pass.

Function Application (Figure 11). When checking applications $f \overset{\pi}{\cdot} a$ 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 rule for erased lambdas, runid applications are well typed if the provided function is a well typed runid function.

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

Fig. 11. Typing rules for application.

Data Eliminators (Figure 12). The eliminator $\mathtt{elNat}\, n\, P\, b\, (m\, p.\, c)$ 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 $\texttt{suc}\, m$ case. We expose the value of m as well as the induction hypothesis $p : P \overset{\omega}{\cdot} m$ by extending the context.

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

$$\frac{\vdash a \overset{\sigma}{:} \texttt{List}\ \ A \qquad \vdash P \overset{0}{:} (y \overset{\pi}{:} \texttt{List}\ \ A) \to \texttt{Set}_i \qquad \vdash b_n \overset{\sigma}{:} P \overset{\pi}{\cdot} \texttt{[]}_l \qquad h \overset{\sigma}{:} A, t \overset{\sigma}{:} \texttt{List}\ \ A, p \overset{\sigma}{:} P \overset{\pi}{\cdot} t \vdash b_c \overset{\sigma}{:} P \overset{\pi}{\cdot} (\texttt{cons}_l\, h\, t)}{\vdash \texttt{elList}\, a\, P\, b_n\, (h\, t\, p.\, b_c) \overset{\sigma}{:} P \overset{\pi}{\cdot} a}$$

$$\frac{\vdash a \overset{\sigma}{:} \texttt{Vec}\, A\, n \qquad \vdash P \overset{0}{:} (m \overset{0}{:} \texttt{Nat}) \to (y \overset{\pi}{:} \texttt{Vec}\, A\, m) \to \texttt{Set}_i \qquad \vdash b_n \overset{\sigma}{:} P \overset{0}{\cdot} \texttt{z} \overset{\pi}{\cdot} \texttt{[]}_v \qquad m \overset{0}{:} \texttt{Nat}, h \overset{\sigma}{:} A, t \overset{\sigma}{:} \texttt{Vec}\, A\, m \vdash b_c \overset{\sigma}{:} P \overset{0}{\cdot} (\texttt{suc}\, m) \overset{\pi}{\cdot} (\texttt{cons}_v\, h\, m\, t)}{\vdash \texttt{elVec}\, a\, P\, b_n\, (h\, m\, t\, p.\, b_c) \overset{\sigma}{:} P \overset{0}{\cdot} n \overset{\pi}{\cdot} a}$$

Fig. 12. Typing rules for the data eliminators.

Runid Data Eliminators (Figure 13). The rule for $\texttt{elNat}_r\, n\, P\, b\, (m\, p.\, c)$ ensures that the output is run-time equivalent to the input in the base cases (like $\texttt{z}$) and in the inductive cases (like $\texttt{suc}\, m$) — where one assumes the subterm m matches the corresponding recursive result p, by extending the assumption context Δ. As an example, such an extension to the assumption context enables us to verify that $\texttt{listToVec} : (x : \texttt{List}\ \ A) \to_r \texttt{Vec}\, A\, (\texttt{length} \overset{\omega}{\cdot} x)$, where

$$\begin{aligned}
\texttt{length} &:= \lambda(x \overset{\omega}{:} \texttt{List}\ \ A).\, \texttt{elList}\ \ x\, (\lambda(l \overset{\omega}{:} \texttt{List}\ \ A).\, \texttt{Nat}) \\
&\qquad\qquad\qquad\qquad\qquad\quad \texttt{z}\, (h\, t\, p.\, \texttt{suc}\, p) \\
\texttt{listToVec} &:= \lambda_r(x : \texttt{List}\ \ A).\, \texttt{elList}_r\ \ x\, (\lambda(l \overset{\omega}{:} \texttt{List}\ \ A).\, \texttt{Vec}\, A\, (\texttt{length} \overset{\omega}{\cdot} l)) \\
&\qquad\qquad\qquad\qquad\qquad\quad \texttt{[]}_v\, (h\, t\, p.\, \texttt{cons}_v\, h\, (\texttt{length} \overset{\omega}{\cdot} t)\, p)
\end{aligned}$$

3 Denotational Semantics of Run-time Equivalence

Our integration of run-time identity functions into the type system hinges on our notion of run-time equivalence, two syntactic code fragments being "equal" at run-time — both in the sense of relating results after erasure (structural) and relating runid computations to their input (extensional). To prove the validity of replacing runid functions by the identity function when compiling expressions,

$$\frac{\begin{array}{c}\vdash \mathtt{elNat}\, n\, P\, b_z\, (m\, p.\, b_s) \overset{\sigma}{:} P \overset{\pi}{\cdot} n \qquad \vdash b_z \sim_r \mathtt{z} : P \overset{\pi}{\cdot} \mathtt{z} \sim_r \mathtt{Nat} \\ m \overset{\sigma}{:} \mathtt{Nat}, p \overset{\sigma}{:} P \overset{\pi}{\cdot} m; p \sim_r m \vdash b_s \sim_r \mathtt{suc}\, m : P \overset{\pi}{\cdot} \mathtt{suc}\, m \sim_r \mathtt{Nat}\end{array}}{\vdash \mathtt{elNat}_r\, n\, P\, b_z\, (m\, p.\, b_s) \overset{\sigma}{:} P \overset{\pi}{\cdot} n}$$

$$\frac{\begin{array}{c}\vdash \mathtt{elList}\, l\, P\, b_n\, (h\, t\, p.\, b_c) \overset{\sigma}{:} P \overset{\omega}{\cdot} a \qquad \vdash b_n \sim_r []_l : P \overset{\omega}{\cdot} []_l \sim_r \mathtt{List}\ A \\ h \overset{\omega}{:} A, t \overset{\omega}{:} \mathtt{List}\ A, p \overset{\omega}{:} P \overset{\omega}{\cdot} t; p \sim_r t \vdash b_c \sim_r \mathtt{cons}_l\, h\, t : P \overset{\omega}{\cdot} \mathtt{cons}_l\, h\, t \sim_r \mathtt{List}\ A\end{array}}{\vdash \mathtt{elList}\, l\, P\, b_n\, (h\, t\, p.\, b_c) \overset{\sigma}{:} P \overset{\omega}{\cdot} as}$$

$$\frac{\begin{array}{c}\vdash \mathtt{elVec}\, a\, P\, b_n\, (h\, m\, t\, p.\, b_c) \overset{\sigma}{:} P \overset{\mathtt{z}}{\cdot} n \overset{\pi}{\cdot} a \qquad \vdash b_n \sim_r []_v : P \overset{0}{\cdot} \mathtt{z} \overset{\pi}{\cdot} []_v \sim_r \mathtt{Vec}\, A\, \mathtt{z} \\ n \overset{0}{:} \mathtt{Nat}, h \overset{\sigma}{:} A, t \overset{\sigma}{:} \mathtt{Vec}\, A\, n, p \overset{\sigma}{:} P \overset{0}{\cdot} n \overset{\pi}{\cdot} t; p \sim_r t \\ \vdash b_c \sim_r \mathtt{cons}_v\, h\, n\, t : P \overset{0}{\cdot} n \overset{\pi}{\cdot} \mathtt{cons}_v\, h\, n\, t \sim_r \mathtt{Vec}\, A\, (\mathtt{suc}\, n)\end{array}}{\vdash \mathtt{elVec}_r\, v\, P\, b_n\, (h\, k\, t\, p.\, b_c) \overset{\sigma}{:} P \overset{0}{\cdot} n \overset{\pi}{\cdot} v}$$

Fig. 13. Typing rules for runid data eliminators.

we need to justify the validity of this syntactic judgement. To this end we give a denotational semantics, a semantic equality judgement $\Gamma =_\Delta \Gamma' \vDash a = b : A = B$ that tells us that two terms are equal at run-time.

More concretely, we define a semantic domain D which we interpret terms into and establish a partial equivalence relation (PER) $a_v \approx b_b : \mathcal{A}$ on values in this domain — based on the model described by Abel in his work on normalization by evaluation (NbE) [1]. Unlike NbE we do not reify back into the original syntax, as we are only interested in equality in the domain D.

3.1 Erasure

We define erasure $\downarrow \cdot$ 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. The function is undefined for explicitly erased terms, e.g. $\downarrow x$ if $x \overset{0}{:} A \in \Gamma$.

This erasure function does *not* treat runid functions in any special way, they are treated precisely as regular (non-erased) functions. The full definition of the erasure function is given in Figure 14.

We define erasure on contexts $\downarrow \Gamma$ to remove all erased variables and apply erasure to all the remaining types. Erasure then satisfies the useful property that it preserves typing:

Lemma 1. *Erasure on well typed terms is well defined and produces again a well typed term: if* $\Gamma \vdash a \overset{\omega}{:} A$ *then* $\downarrow \Gamma \vdash \downarrow a \overset{\omega}{:} \downarrow A$.

Proof. By induction on the typing rules.

In other variants of QTT, erasure might not preserve typing since erased types can still be used in terms (e.g. as type annotations on lambda expressions).

$$
\begin{array}{lll}
\downarrow x & = x & \text{(Variables)} \\
\downarrow(a \overset{0}{:} A) \to B & = \downarrow B & \text{(Types)} \\
\downarrow(a \overset{\omega}{:} A) \to B & = (a :\downarrow A) \to \downarrow B & \\
\downarrow(x : A) \to_r B & = (a :\downarrow A) \to_r \downarrow B & \\
\downarrow\texttt{Nat} & = \texttt{Nat} & \\
\downarrow\texttt{List}\ A & = \texttt{List}\ \downarrow A & \\
\downarrow\texttt{Vec}\, A\, n & = \texttt{List}\ \downarrow A & \\
\downarrow\texttt{Set}_i & = \texttt{Set}_i & \\
\downarrow\lambda(a \overset{0}{:} A).\, b & = \downarrow b & \text{(Constructors)} \\
\downarrow\lambda(a \overset{\omega}{:} A).\, b & = \lambda(a :\downarrow A).\ \downarrow b & \\
\downarrow\lambda_r(a : A).\, b & = \ \lambda_r(a :\downarrow A).\ \downarrow b & \\
\downarrow[]_l & = []_l & \\
\downarrow\texttt{cons}_l\, h\, t & = \ \texttt{cons}_l\ \downarrow h\ \downarrow t & \\
\downarrow[]_v & = []_l & \\
\downarrow\texttt{cons}_v\, h\, n\, t & = \ \texttt{cons}_l\ \downarrow h\ \downarrow t & \\
\downarrow\texttt{z} & = \texttt{z} & \\
\downarrow\texttt{suc}\, n & = \texttt{suc}\ \downarrow n & \\
\downarrow f \overset{0}{\cdot} a & = \downarrow f & \text{(Applications)} \\
\downarrow f \overset{\omega}{\cdot} a & = \downarrow f \cdot \downarrow a & \\
\downarrow f \cdot_r a & = \downarrow f \cdot_r \downarrow a & \\
\downarrow\texttt{elList}\, a\, P\, b\, (h\, t\, p.\, c) & = \texttt{elList}\ \downarrow a\ \downarrow P\ \downarrow b\, (h\, t\, p.\ \downarrow c) & \text{(Eliminations)} \\
\downarrow\texttt{elList}_r\, a\, P\, b\, (h\, t\, p.\, c) & = \texttt{elList}\ \downarrow a\ \downarrow P\ \downarrow b\, (h\, t\, p.\ \downarrow c) & \\
\downarrow\texttt{elVec}\, a\, P\, b\, (h\, n\, t\, p.\, c) & = \texttt{elList}\ \downarrow a\ \downarrow P\ \downarrow b\, (h\, t\, p.\ \downarrow c) & \\
\downarrow\texttt{elVec}_r\, a\, P\, b\, (h\, n\, t\, p.\, c) & = \texttt{elList}\ \downarrow a\ \downarrow P\ \downarrow b\, (h\, t\, p.\ \downarrow c) & \\
\downarrow\texttt{elNat}\, a\, P\, b\, (m\, p.\, c) & = \texttt{elNat}\ \downarrow a\ \downarrow P\ \downarrow b\, (m\, p.\ \downarrow c) & \\
\downarrow\texttt{elNat}_r\, a\, P\, b\, (m\, p.\, c) & = \texttt{elNat}\ \downarrow a\ \downarrow P\ \downarrow b\, (m\, p.\ \downarrow c) &
\end{array}
$$

Fig. 14. Definition of erasure on terms of our language.

3.2 Domain Values

The domain D is defined by the equations in Figure 15. It consists of two parts: D_V contains values that terms map to and 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 $a \mapsto b : D \to D$, a pair $(a, b) : D \times D$ or an empty list symbol $[]$. Accordingly, a type code $d_t : D_T$ is either: $\texttt{Nat}_D$, a list type code $\texttt{List}_D(a)$, a dependent function type code $\texttt{Fun}_D(a, b)$, a runid function type code $\texttt{Runid}_D(a, b)$, or a universe type code $\texttt{Set}_D(i)$ (with a universe level i).

Following Abel [1], the domain D itself is constructed as an applicative structure where functions $D \to D$ are represented as defunctionalized closures, which

$$
\begin{array}{lll}
D & = & D_V \quad \cup \quad D_T \\
D_V & = & \mathbb{N} \quad \cup \quad D \to D \quad \cup \quad D \times D \quad \cup \quad \{[]\} \\
D_T & = & \{\mathtt{Nat}_D\} \ \cup \ \{\mathtt{List}_D(a) \mid a \in D_T\} \ \cup \ \{\mathtt{Set}_D(i) \mid i \in \mathbb{N}\} \\
 & & \{\mathtt{Fun}_D(a,b) \mid a \in D_T, b \in D \to D_T\} \ \cup \ \{\mathtt{Runid}_D(a,b) \mid a \in D_T, b \in D \to D_T\}
\end{array}
$$

Fig. 15. Values in D

is sufficient to guarantee that our set of equations has a fixed point. We present these closures as regular functions for ease of reading.

3.3 Evaluating Terms into Domain Values

We define an evaluation function $(\!|a|\!)_\gamma$ from terms into our domain D in Figure 16. Here γ is an environment mapping the free variables in a to their domain values. We intend to only apply evaluation after applying erasure, so we only define it for term constructors that can still be present after erasure.

$$
\begin{array}{lll}
(\!|x|\!)_\gamma & = \gamma(x) & \text{(Variables)} \\
(\!|(a : A) \to B|\!)_\gamma & = \mathtt{Fun}_D((\!|A|\!)_\gamma, a_v \mapsto (\!|B|\!)_{\gamma[a \mapsto a_v]}) & \text{(Types)} \\
(\!|(a : A) \to_r B|\!)_\gamma & = \mathtt{Runid}_D((\!|A|\!)_\gamma, a_v \mapsto (\!|B|\!)_{\gamma[a \mapsto a_v]}) & \\
(\!|\mathtt{List}\ A|\!)_\gamma & = \mathtt{List}_D((\!|A|\!)_\gamma) & \\
(\!|\mathtt{Nat}|\!)_\gamma & = \mathtt{Nat}_D & \\
(\!|\mathtt{Set}_i|\!)_\gamma & = \mathtt{Set}_D(i) & \\
(\!|\lambda(a : A).\, b|\!)_\gamma & = a_v \mapsto (\!|b|\!)_{\gamma[a \mapsto a_v]} & \text{(Constructors)} \\
(\!|\lambda_r(a : A).\, b|\!)_\gamma & = a_v \mapsto (\!|b|\!)_{\gamma[a \mapsto a_v]} & \\
(\!|\mathtt{z}|\!)_\gamma & = 0 & \\
(\!|\mathtt{suc}\, n|\!)_\gamma & = 1 + (\!|n|\!)_\gamma & \\
(\!|[]_l|\!)_\gamma & = [] & \\
(\!|\mathtt{cons}_l\, a\, b|\!)_\gamma & = ((\!|h|\!)_\gamma,\ (\!|t|\!)_\gamma) & \\
(\!|f \cdot a|\!)_\gamma & = (\!|f|\!)_\gamma((\!|a|\!)_\gamma) & \text{(Eliminations)} \\
(\!|f \cdot_r a|\!)_\gamma & = (\!|f|\!)_\gamma((\!|a|\!)_\gamma) & \\
(\!|\mathtt{elNat}\, a\ \_\ b\, (m\, p.\, c)|\!)_\gamma & = \mathtt{rec}_{\mathbb{N}}((\!|a|\!)_\gamma, (\!|b|\!)_\gamma, (k, r) \mapsto (\!|c|\!)_{\gamma[m \mapsto k, p \mapsto r]}) & \\
 & \text{where } \mathtt{rec}_{\mathbb{N}}(0, b, i) = b & \\
 & \qquad \mathtt{rec}_{\mathbb{N}}(1 + m, b, i) = i(m, \mathtt{rec}_{\mathbb{N}}(m, b, i)) & \\
(\!|\mathtt{elList}\, xs\ \_\ b\, (c)\ .\ \gamma)|\!) & = \mathtt{rec}_L((\!|xs|\!)_\gamma, (\!|b|\!)_\gamma, (h, t, r) \mapsto (\!|c|\!)_{\gamma[a \mapsto h, as \mapsto t, p \mapsto r]}) & \\
 & \text{where } \mathtt{rec}_L([], b, i) = b & \\
 & \qquad \mathtt{rec}_L((h, t), b, i) = i(h, t, \mathtt{rec}_L(t, b, i)) &
\end{array}
$$

Fig. 16. Evaluation of terms into domain values.

Evaluating Variables. The evaluation function fully evaluates terms to the corresponding domain values. To evaluate open terms (i.e. terms with free variables), we look up the variable in the environment γ.

Evaluating Constructors. Lambdas are evaluated into semantic functions from a value a_v to some definition given by the evaluation of the lambda body. The body extends the environment to bind the variable a to the provided argument value a_v.

Evaluating Eliminations. Lambda application evaluates to function application in the domain. Meanwhile, eliminators on recursive data are defined by two recursive helper functions $\texttt{rec}_{\mathbb{N}}$ and $\texttt{rec}_L$ on the domain. 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 primitive recursion operations.

Evaluating Types as Terms. Since we can encounter types in terms we need to map them to a domain value during evaluation. Evaluating syntactic types gives us an *encoding* of types in our semantic domain, *not* a semantic type. In the definition of our PER model in the next subsection we refer to the semantic types as type *values* — to avoid confusion we will refer to the result of evaluating terms type *codes* instead. 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. Hence a function type code $\texttt{Fun}_D(A, F)$ is the type code for its argument type $A \in D_T$ together with a semantic function from its argument to its result type code $F : D \to D_T$.

3.4 Partial Equivalence Relation

Now that we have defined operations for creating domain values a_v, b_v we need some mechanism to relate values in accordance with the type they belong to. This is done by representing semantic types as PERs $\mathscr{A} \subseteq D \times D$ on the domain. We will write $a_v \approx b_v : \mathscr{A}$ for $(a_v, b_v) \in \mathscr{A}$. 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 in an inductive fashion in Figure 17, building them "from the ground up" following Abel [1].

$\Pi(\mathscr{A}, \mathscr{F})$ denotes a semantic dependent function type where $\mathscr{A}$ is the argument type and $\mathscr{F} : \mathscr{A} \to \mathtt{Per}$ the type family. It is defined through function extensionality: two functions $f, f' \in D \to D$ are related by $\Pi(\mathscr{A}, \mathscr{F})$ if they map related inputs to related outputs. The semantic type of runid functions $\Pi_r(\mathscr{A}, \mathscr{F})$ is defined similarly, except that function outputs are related to their inputs — which by transitivity and symmetry imply that function outputs are also related to each other as in regular functions. Inductive constructors are related inductively by congruence. The base cases are related to themselves (e.g. $0 \approx 0 : \mathscr{N}at$) and the inductive cases are related if their sub-values are.

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

$$\frac{}{0 \approx 0 : \mathscr{N}at} \qquad \frac{n \approx m : \mathscr{N}at}{1 + n \approx 1 + m : \mathscr{N}at} \qquad \frac{}{[] \approx [] : \mathscr{L}ist(\mathscr{A})}$$

$$\frac{h \approx h' : \mathscr{A} \qquad t \approx t' : \mathscr{L}ist(\mathscr{A})}{(h, t) \approx (h', t') : \mathscr{L}ist(\mathscr{A})} \qquad \frac{}{\mathtt{Nat}_D \approx \mathtt{Nat}_D : \mathscr{S}et_0}$$

$$\frac{A \approx A' : \mathscr{S}et_i \qquad \forall a \approx a' : [A]. B(a) \approx B'(a') : \mathscr{S}et_j}{\mathtt{Fun}_D(A, B) \approx \mathtt{Fun}_D(A', B') : \mathscr{S}et_{\max(i,j)}} \qquad \frac{}{\mathtt{Set}_D(i) \approx \mathtt{Set}_D(i) : \mathscr{S}et_{i+1}}$$

Fig. 17. Semantic types are defined inductively as PERs on domain values.

Different type codes may denote the same semantic type. Therefore we define the PERs on type codes (semantic universes) inductively. When relating a function type code $\mathtt{Fun}_D(A, B)$ we need to interpret the type code A into a semantic type. We interpret type codes as semantic types $[\_]$ as follows:

$$\begin{array}{ll} [\mathtt{Nat}_D] & = \mathscr{N}at \\ [\mathtt{List}_D(A)] & = \mathscr{L}ist([A]) \\ [\mathtt{Fun}_D(A, B)] & = \Pi([A], a \mapsto [B(a)]) \\ [\mathtt{Runid}_D(A, B)] & = \Pi_r([A], a \mapsto [B(a)]) \\ [\mathtt{Set}_D(i)] & = \mathscr{S}et_i \end{array}$$

3.5 Defining a Semantic Run-time Equality

We now 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. To interpret a term we first apply erasure and then evaluate.

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

Types are interpreted into partial equivalence relations, giving a semantic account of which values belong to a given semantic type and how we can compare two values of that type. To interpret a syntactic type we first apply erasure, then evaluate to a type code, then lift to a semantic type.

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

In order to interpret terms we need to provide environments for the variables present in the left and right terms, subject to the assumed equivalences stated in the assumption context. To that end we interpret contexts and their assumptions simultaneously (Figure 18). We ignore erased variables (S-$EXT0L/R$), extend environments with equivalent values (S-EXT) and strengthen existing environments when the assumption context requires an equivalence of variable mappings already introduced to the environment (S-STR).

$$\frac{}{\emptyset \approx \emptyset : [\![\emptyset \sim_r \emptyset; \emptyset]\!]}\text{S-}NIL$$

$$\frac{\gamma \approx \gamma' : [\![\Gamma \sim_r \Gamma'; \Delta]\!] \qquad a \approx a' : [\![A]\!]_\gamma}{\gamma[x \mapsto a] \approx \gamma'[x' \mapsto a'] : [\![\Gamma, x \overset{\omega}{:} A \sim_r \Gamma', x' \overset{\omega}{:} A; \Delta, x \sim_r x']\!]}\text{S-}EXT$$

$$\frac{x \overset{\omega}{:} A \in \Gamma \qquad \gamma(x) \approx \gamma'(x') : [\![A]\!]_\gamma \qquad \gamma \approx \gamma' : [\![\Gamma \sim_r \Gamma'; \Delta]\!]}{\gamma \approx \gamma' : [\![\Gamma \sim_r \Gamma'; \Delta, x \sim_r x']\!]}\text{S-}STR$$

$$\frac{\gamma \approx \gamma' : [\![\Gamma \sim_r \Gamma'; \Delta]\!]}{\gamma \approx \gamma' : [\![\Gamma, x \overset{0}{:} A \sim_r \Gamma'; \Delta]\!]}\text{S-}EXT0L \qquad \frac{\gamma \approx \gamma' : [\![\Gamma \sim_r \Gamma'; \Delta]\!]}{\gamma \approx \gamma' : [\![\Gamma \sim_r \Gamma', x' \overset{0}{:} A'; \Delta]\!]}\text{S-}EXT0R$$

Fig. 18. Triples $\Gamma \sim_r \Gamma'; \Delta$ of two contexts Γ and Γ' and an assumption context Δ' are interpreted as a logical relation on environments.

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

Definition 1. *Terms are semantically run-time equivalent if for all related environments they have interpreted to equivalent values. Environments and values are determined equivalent by the PER obtained from interpreting the context and syntactic type respectively.*

$$\Gamma =_\Delta \Gamma' \vDash a = b : A = B \iff \forall \gamma \approx \gamma' : [\![\Gamma \sim_r \Gamma'; \Delta]\!].\ [\![a]\!]_\gamma \approx [\![b]\!]_{\gamma'} : [\![A]\!]_\gamma$$

In this definition, we arbitrarily choose $[\![A]\!]_\gamma$ to relate the two values rather than $[\![B]\!]_\gamma$. In the examples we have considered so far, A and B are always the same up to erasure and hence the choice does not matter. We conjecture that for well-typed terms a and b we have that $\vdash a \sim_r b : A \sim_r B$ implies that $\vdash A \sim_r B : \mathtt{Set}_i \sim_r \mathtt{Set}_j$ and hence A and B always denote the same relation, but we have not yet proven this formally.

This in turn allows us to define a semantic typing judgement:

Definition 2. *A term is semantically well typed if it is semantically related to itself (where $\Gamma^{\approx}$ relates each variable in Γ to itself).*

$$\Gamma \vDash a : A \iff \Gamma =_{\Gamma^{\approx}} \Gamma \vDash a = a : A = A$$

4 Syntactically Related Terms Are Semantically Related

Now that we have established a semantic logical relation to define run-time equivalence, we prove the "fundamental property" of this logical relation: our syntactic relation entails the semantic relation. In other words the syntactic notion of run-time equivalence implies semantic equivalence in the PER model.

Theorem 1 (Fundamental property). *Syntactically well typed terms are semantically related to themselves, and syntactically related (well typed) terms are semantically related.*

$$\Gamma \vdash a \overset{\sigma}{:} A \implies \Gamma \vDash a : A$$
$$\Gamma \vdash b \overset{\sigma}{:} B \implies \Gamma \vDash b : B$$
$$\Gamma \sim_r \Gamma' \,;\, \Delta \vdash a \sim_r b : A \sim_r B \implies \Gamma =_{\Delta} \Gamma' \vDash a = b : A = B$$

The proof proceeds by mutual induction on the rules of the run-time equivalence relation and the typing rules. We lack the space to list all cases exhaustively in this paper. However, they can be grouped up and proven with the strategies as follows:

1. The cases for the PER rules (symmetry and transitivity) and congruence rules follow directly by induction.
2. Eliminators are interpreted as the output of a function and thus follow by induction and function extensionality in the domain. Validity of motives ensures that eliminators produce well typed families in our semantics.
3. The rules for terms with erasure (e.g. erased lambdas and erased applications) follow directly from their induction hypotheses by the fact that our semantics is defined via erasure.
4. The rules for runid functions carry over because their semantic interpretation ignores the runid marker. The inductive step for runid eliminators (e.g. the eliminator on `Nat`) relies on the induction hypothesis corresponding to the run-time equivalence hypotheses in their typing rules.

Note that for runid terms the typing rules enforce properties in terms of run-time equivalences (e.g. the rules for the runid data eliminators containing run-time equivalence judgements on the branches). This is why we need to rely on the well-typedness of terms in our syntactic equivalence relation — as it allows us to obtain these properties in our proof.

We spell out the details for three of the more interesting cases, as representative examples of how the proof operates.

Case 1 (runid lambda).

$$\frac{}{\vdash \lambda_r(x : A).\, b \sim_r \lambda_r(x : A).\, x \;:\; (x : A) \to_r B \sim_r (x : A) \to_r A}$$

Proof. Since the terms evaluate to semantic functions we need to show that they map related inputs to related outputs. We can prove this by using the induction hypothesis corresponding to the second condition in the typing rule for runid lambdas:

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

Case 2 (runid application).

$$\frac{\vdash a \sim_r a' \;:\; A \sim_r A}{\vdash f \cdot_r a \sim_r a' \;:\; B \sim_r A}$$

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

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

By induction on the typing rules we see that f is a well typed runid function and thus semantically $[\![f]\!]_\gamma([\![a]\!]_\gamma) \approx [\![a]\!]_\gamma : [\![A]\!]_\gamma$. Moreover, we know that $[\![a]\!]_\gamma \approx [\![a']\!]_{\gamma'} : [\![A]\!]_\gamma$ holds by induction on the assumption. Hence the conclusion follows by transitivity.

Case 3 (Runid Nat eliminators).

$$\frac{\vdash x \sim_r x' \;:\; \mathtt{Nat} \sim_r \mathtt{Nat}}{\vdash \mathtt{elNat}_r\, x\, P\, b\, (m\, p.\, c) \sim_r x' \;:\; P \overset{\omega}{\cdot} x \sim_r \mathtt{Nat}}$$

Proof. We operate by induction on the semantic value of the scrutinee x, which evaluates to some natural number value m. Once again we rely on the typing rule:

$$\frac{\begin{array}{c}\vdash \mathtt{elNat}\, n\, P\, b_z\, (m\, p.\, b_s) \overset{\sigma}{:} P \overset{\pi}{\cdot} n \qquad \vdash b_z \sim_r \mathtt{z} \;:\; P \overset{\pi}{\cdot} \mathtt{z} \sim_r \mathtt{Nat} \\ m \overset{\sigma}{:} \mathtt{Nat}, p \overset{\sigma}{:} P \overset{\pi}{\cdot} m; p \sim_r m \vdash b_s \sim_r \mathtt{suc}\, m \;:\; P \overset{\pi}{\cdot} \mathtt{suc}\, m \sim_r \mathtt{Nat}\end{array}}{\vdash \mathtt{elNat}_r\, n\, P\, b_z\, (m\, p.\, b_s) \overset{\sigma}{:} P \overset{\pi}{\cdot} n}$$

The conclusion now follows by induction on the number m: if it is 0 then we apply the induction hypothesis for the base branch, and if it is a successor we apply the induction hypothesis for the inductive branch.

5 Related Work

Quantitative Type Theory. Our work builds heavily on Quantitative Type Theory (QTT) by Atkey [2]. QTT allows one to specify strict usage conditions on

variables; whether they are used zero times (i.e. used only at compile time), once (linearly), twice (etc), or an unrestricted number of times. Since we only support the quantities 0 and ω, we can simplify the type theory. On the other hand, the addition of a type of run-time identity functions extends the type theory in a way orthogonal to supporting more quantities. We expect our work to be compatible with having more quantities, though the details remain to be worked out.

Agda2HS. Agda2HS [4] provides a translation from a subset of Agda to Haskell which is very similar to the erasure operation defined in this paper. It also provides a way to annotate runid functions with the "transparent" keyword, which is checked by comparing the left- and right-hand sides of each function clause after erasure. However, it does not reflect this information in the type system and thus does not allow downstream optimizations such as higher order runid functions.

Ornaments. Ornaments [7] 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). While ornaments are mostly concerned with identifying the structure of types as being ornaments of other types, our work in this paper is more aimed at exploiting the fact that two types have the same representation at run-time.

Custom Representations. Theocharis et al. [12] 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 run-time 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, but can still be tedious. In contrast, our approach only requires annotating types with their run-time representation and annotating runid identity functions, without any additional proving necessary.

Proof Irrelevance. Many dependently typed language such as Rocq allow for erasure of proofs by having a separate run-time irrelevant universe `Prop`, separating types of proofs from other types [6]. So a compiler could simply disregard members of `Prop`. However, this approach fails to erase type-level indices other than proofs. 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 from `Prop` to `Type`.

Ghost Type Theory. Winterhalter [14] tries to alleviate the shortcomings of `Prop` through a universe of "ghost" types, which behave closer to run-time irrelevance.

This allows for more granularity by discarding impossible branches, enabling the vector example we gave for Rocq. As ghost types and the 0 quantity from QTT are really two sides of the same coin, we expect it to be possible to combine our approach to ghost type theory as well. However, we leave the details to future work.

Identity Function Detection in Compiler Backends. Some compilers already perform identity optimisations for commonplace shapes. For example, Idris 2 recognizes identity functions on list-shaped things [10], among others. However this does not give the programmer any mechanism to assure that this optimization will kick in, nor to express the requirement that a given input function should be a run-time identity.

Boitel [3] shows an approach to eliminating identity functions in OCaml for data types with equivalent memory representation. This shows that even non-dependently typed languages could benefit from having a dedicated type of run-time identity functions.

Two Level Type-Theory. Two level type-theory (2LTT) has been used to describe a staged compilation framework for dependent types, enabling the programmer to define their own compiler optimizations by splitting the language into an object level and a metaprogram level [5]. This is similar to the distinction between a run-time and compile-time level in QTT that we use. However, 2LTT does not enable the programmer to perform our runid optimizations as it takes place in an earlier stage of compilation, so the two approaches are complementary.

6 Conclusion and future work

We have developed a core dependently typed calculus 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 semantics based on partial equivalence relations (PER), and an erasure translation made precise which parts of a program survive compilation. We sketched a proof that syntactically related erased terms are semantically equal after erasure. Together, these results provide a foundation for reasoning about modalities on run-time behaviour in dependent type theory and for optimizing dependently typed programs. The next steps for this work are clear: adding support for inductive families more generally, giving a more principled account of how to deal with erasure of type annotations, providing a full mechanized proof of our fundamental property, and implementing the `@repr` and `@runid` annotations for Agda.

Acknowledgments. This paper based on the Master thesis by the first author, Padilla Cancio[1].

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

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: 33rd ACM/IEEE Symposium on Logic in Computer Science (LICS). ACM (2018). https://doi.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: 15th ACM SIGPLAN International Haskell Symposium. pp. 108–122. ACM (2022). https://doi.org/10.1145/3546189.3549920
5. Kovács, A.: Staged compilation with two-level type theory. Proceedings of the ACM on Programming Languages **6**(ICFP), 540–569 (2022). https://doi.org/10.1145/3547641
6. Letouzey, P.: A new extraction for Coq. In: Second International Workshop on Types for Proofs and Programs, Selected Papers. Lecture Notes in Computer Science, vol. 2646. Springer (2002). https://doi.org/10.1007/3-540-39185-1_12
7. McBride, C.: Ornamental algebras, algebraic ornaments (2011), https://personal.cis.strath.ac.uk/conor.mcbride/pub/OAAO/LitOrn.pdf, (unpublished)
8. McBride, C., McKinna, J.: The view from the left. Journal of Functional Programming **14**(1), 69–111 (2004). https://doi.org/10.1017/S0956796803004829
9. New, M.S., Licata, D.R., Ahmed, A.: Gradual type theory. Proceedings of the ACM on Programming Languages **3**(POPL) (2019). https://doi.org/10.1145/3290328, https://doi.org/10.1145/3290328
10. Stafford, Z.: Make CONS, NIL, JUST and NOTHING constructors have uniform names (2025), https://github.com/idris-lang/Idris2/pull/3486, accessed 13-08-2025
11. The Agda Development Team: The Agda programming language 2.8.0, https://hackage.haskell.org/package/Agda-2.8.0
12. Theocharis, C., Brady, E.: Custom representations of inductive families. In: 26th International Symposium on Trends in Functional Programming, Revised Selected Papers. Springer Nature (2025). https://doi.org/10.1007/978-3-031-99751-8_13
13. Wadler, P.: Views: A way for pattern matching to cohabit with data abstraction. In: 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL). pp. 307–313 (1987). https://doi.org/10.1145/41625.41653
14. 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