

Erasure and Polymorphism in Pure Type Systems

Nathan Mishra Linger and Tim Sheard

Portland State University
{`rlinger, sheard`}@cs.pdx.edu

Abstract. We introduce *Erasure Pure Type Systems*, an extension to Pure Type Systems with an erasure semantics centered around a type constructor \forall indicating parametric polymorphism. The erasure phase is guided by lightweight program annotations. The typing rules guarantee that well-typed programs obey a phase distinction between erasable (compile-time) and non-erasable (run-time) terms.

The erasability of an expression depends only on how its value is *used* in the rest of the program. Despite this simple observation, most languages treat erasability as an intrinsic property of expressions, leading to code duplication problems. Our approach overcomes this deficiency by treating erasability extrinsically.

Because the execution model of EPTS generalizes the familiar notions of type erasure and parametric polymorphism, we believe functional programmers will find it quite natural to program in such a setting.

1 Background and Motivation

Statically typed programming languages have evolved ever more expressive type systems. The drive towards increased expressiveness inevitably leads to dependent types, a proven technology for verifying strong correctness properties of real programs [15, 9, 5, 8]. For this reason, researchers have long sought practical ways to include dependent types in programming languages.

Heavy use of the expressive power of dependent types involves the embedding of proofs of program properties into the program text itself. Often, such proofs play no essential part in the execution of the program. They are necessary only as evidence used by the type-checker. Because these proofs can be quite large, an erasure semantics is critical for practical implementation of a dependently typed programming language.

However, proofs are not the only erasable parts of a program. Any time a program exhibits *parametric polymorphism* (whether it be polymorphism over proofs, types, numbers, or any other type of value) there are portions of the program that should be erased. One thesis of this paper is that parametric polymorphism can be understood entirely in terms of erasability.

1.1 Erasability is extrinsic rather than intrinsic

Which parts of a program¹ may be erased in an erasure-semantics? Our investigation of erasure semantics is grounded in a simple observation: Erasability of a program expression is not a property of the expression itself but rather a property of the context in which we find it. Erasability of a program expression is determined not by what it *is*, but by how it is *used*. In other words, erasability is an *extrinsic* rather than an *intrinsic* property. We give several examples demonstrating this principle.

Type annotations. The domain annotation A in the β rule,

$$(\lambda x:A. M) N \rightarrow_{\beta} M[N/x]$$

is simply discarded. For this reason, we may safely erase the domain annotations of all λ -abstractions in a program without changing their computational behavior. In this case, the context in which A appears determines its erasability.

Dummy λ -binders. The erasure of domain annotations may cause some λ -binders to become superfluous. Consider the term $(\lambda\alpha:\text{Type}. \lambda x:\alpha. x) Nat\ 5$. After erasing type annotations, we are left with $(\underline{\lambda\alpha}. \lambda x. x) \underline{Nat}\ 5$, in which the binder $\lambda\alpha$ is superfluous because α no longer appears anywhere in its scope. For any such dummy binder λx , the resulting specialized β rule

$$(\lambda x. M) N \rightarrow_{\beta} M \quad \text{if } x \notin FV(M)$$

discards both the dummy binder and the argument which it would otherwise bind to x . Therefore we may erase both the binding site λx and any argument N at an application site to which this λ -abstraction may flow during program execution. By this reasoning, we may erase the underlined portions of our previous example term, resulting in $(\lambda x. x)\ 5$.

However, other λ -abstractions may flow to some of those same application sites. We should not erase the argument N at an application site² $M@N$ unless every λ -abstraction that may flow to be the value of M has a dummy binder. In general, the “may-flow-to” relation induces a bipartite graph (see Figure 1). In order to decide if a given λ -binder or $@$ -argument may be safely erased, we must analyze its entire connected component (CC) in this $\lambda/@$ graph.

In this type of erasure step, the *usage* of a term determines its erasability. The (local) erasability of a binder λx depends on how x is (or is not) used in its scope. The erasability of an argument N depends on its context — whether the function that is applied to it always ends up being a λ -abstraction with a dummy binder.

¹ For our purposes here, a program is a term in a typed λ -calculus.

² We sometimes write $@$ for application in order to have a more tangible notation than mere juxtaposition.

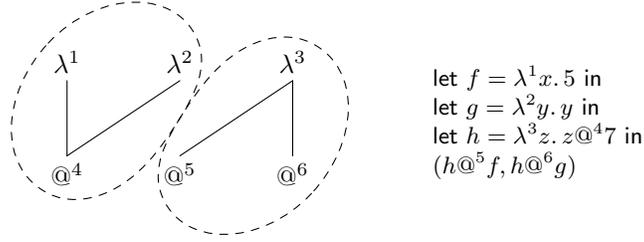


Fig. 1. The $\lambda/@$ graph induced by the “may-flow-to” relation of a simple program. The use of y in λ^2 ’s body prevents erasure of both the $@^4$ -argument and the λ^1 -binder.

Cascading Erasure. Erasure of $@$ -arguments may make other λ -binders into dummies, thereby enabling erasure in other CCs of the $\lambda/@$ graph. Consider the following family of identity functions.

let $id_0 = \lambda a:s. \lambda x:a. x$ in	let $id_0 = \lambda a. \lambda x. x$ in
let $id_1 = \lambda a:s. \lambda x:a. id_0 a x$ in	let $id_1 = \lambda a. \lambda x. id_0 a x$ in
let $id_2 = \lambda a:s. \lambda x:a. id_1 a x$ in	\Rightarrow let $id_2 = \lambda a. \lambda x. id_1 a x$ in
let $id_3 = \lambda a:s. \lambda x:a. id_2 a x$ in	let $id_3 = \lambda a. \lambda x. id_2 a x$ in
...	...

After the initial erasure of domain annotations, a cascading sequence of $\lambda/@$ erasure steps is possible in this program. (Consider the λa binders).

1.2 Intrinsic notions of erasability beget code duplication

Most prior attempts to combine dependent types and erasure semantics treat erasability as an intrinsic property. These attempts may be divided into two categories: erasure first and dependent types first.

Erasure first. Languages in this category start with a commitment to erasure semantics in the form of a syntactic phase distinction whereby types and program values may not depend on each other computationally. Singleton types are then used to simulate dependently typed programming. Examples of this approach include Dependent ML [22], Ω mega [20,19], Applied Type Systems [7], and Haskell with generalized algebraic datatypes [16].

Singleton types are type families $T:I \rightarrow \text{Type}$ for which each type index $i:I$ uniquely determines the one value of type $T i$. For example, the declarations

datakind $Nat\uparrow$: kind	datatype $Nat!$: $Nat\uparrow \rightarrow \text{Type}$
where $Zero\uparrow$: $Nat\uparrow$	where $Zero!$: $Nat!$ $Zero\uparrow$
$Succ\uparrow$: $Nat\uparrow \rightarrow Nat\uparrow$	$Succ!$: $Nat! n \rightarrow Nat! (Succ\uparrow n)$

introduce a singleton type family for the naturals. The `datakind` declaration defines a *copy* of the natural numbers at the type-level. The singleton type $Nat!$ then connects the type-level version $Nat\uparrow$ to the level of run-time expressions.

A singleton type acts as a proxy between run-time and compile-time notions of the same datatype: natural numbers in this case. Whenever a program does case analysis on the value of a singleton type, the type-checker benefits from the same case analysis at the type-level. In this way, dependence of types on values is simulated.

Dependent types first. Languages in this category start with full dependent types. An erasure phase then strips out parts of the program that are irrelevant to its run-time execution. Examples of this approach include Cayenne [3], Coq [1], and Epigram [12, 6].

In Cayenne and Coq, the erasability of a subterm depends on its type. All types, subterms of type `Type`, are erased in Cayenne and Epigram³. Coq’s program extraction mechanism supports erasure of *proofs* as well as types. A proof is distinguished by having a proposition as its type, and propositions are distinguished as terms of type `Prop`. In contrast to the universe `Prop`, Coq has another universe `Set` that is the type of the types of all non-erasable program terms.

Code duplication. Because languages in both categories treat erasability as an intrinsic property of an expression, usually determined by its type, users of these languages are sometimes forced to duplicate definitions of datatypes and functions over them in order to achieve a desired erasure behavior. In the erasure-first approach, programming with singleton types requires duplication of datatype definitions at the “type” and “kind” levels of the type hierarchy, as well as duplication of functions that operate on them. In the dependent-types-first approach, duplication of datatypes is also required if we want values of a particular type to be erased in one part of a program but not in another.

1.3 Methodology and Outline

We treat erasability as a property not of a term itself, but of the context in which it is used. In λ -calculus, functions reify such contexts, so we track erasability as a property of functions by distinguishing between functions that do not depend computationally on their arguments (of type $\forall x:A. B$) and those that might (of type $\exists x:A. B$).

Note that the same A is used in both cases, because erasability is no longer an intrinsic property of x , but rather a property of the (functional) contexts making use of x . In this way we avoid the code duplication problem. We have *one* type A and therefore functions over A can be written *once*.

Pure Type Systems (PTS) are a well-known family of typed λ -calculi that encompass a wide variety of type systems [4]. Most dependently typed languages have a PTS at their core. Therefore PTS is a good setting for studying features of dependently types languages. Section 2 briefly reviews the basics of PTS.

³ Some work on representations of inductive types in Epigram notes that values of type families need not store certain indices that, regardless of their type, are uniquely determined by the value’s data constructor

Section 3 introduces a conservative extension to PTS called Erasure Pure Type Systems (EPTS) supporting the \forall type described above and rules for checking that programs using this type satisfy a *phase distinction*.

Section 4 introduces another PTS variant called Implicit Pure Type Systems (IPTS), that serves as the target language of the erasure phase. This language is very closely related to Miquel’s Implicit Calculus of Constructions [13].

Section 5 introduces our erasure translation from EPTS to IPTS. This operation is the basis for the erasure semantics of EPTS (Section 6). We prove that erasure exhibits properties one would expect: It respects the static and dynamic semantics of programs and *eliminates portions of the source program that do not affect its final value*.

In Sections 7, 8, and 9 we discuss implementation issues, future work, and our conclusions.

2 Pure Type Systems

Pure Type Systems bring organization to type theory [4]. They generalize Barendregt’s λ -cube, which includes such familiar systems as the simply typed λ -calculus, Systems F and F^ω , the Edinburgh Logical Framework, and the Calculus of Constructions.

Pure Type Systems are a family of typed lambda calculi. Each PTS has a specification $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ consisting of a set \mathcal{S} of *sorts* (a.k.a. *universes*), a set $\mathcal{A} \subseteq \mathcal{S}^2$ of *axioms*, and a set $\mathcal{R} \subseteq \mathcal{S}^3$ of *rules*. We assume a fixed specification throughout the development. The syntax of PTS is as follows:

$$M, N, A, B ::= x \mid \lambda x:A. M \mid M N \mid \Pi x:A. B \mid s$$

Note that there is a single syntactic category for types and terms. The metavariable s is used to denote sorts. The typing rules of PTS are parameterized by \mathcal{A} and \mathcal{R} and can be obtained from those of EPTS (which we will discuss shortly) by simply ignoring all erasure annotations.

3 Erasure Pure Type Systems

This section introduces Erasure Pure Type Systems (EPTS), an extension of Pure Type Systems (PTS) with annotations indicating erasable parts of a program. The EPTS type system checks the erasability of the parts so annotated.

Syntax. The syntax of EPTS is that of PTS with erasure annotations added.

$$\begin{array}{ll} \text{(terms)} & M, N, A, B ::= x \mid \lambda^\tau x:A. M \mid M @^\tau N \mid \Pi^\tau x:A. B \mid s \\ \text{(contexts)} & \Gamma, \Delta ::= \varepsilon \mid \Gamma, x:\tau A \\ \text{(times)} & \tau ::= \mathbf{r} \mid \mathbf{c} \end{array}$$

The metavariable τ ranges over erasure annotations. The annotation \mathbf{r} means “run-time”. Syntax with this annotation behaves just as it would in PTS without

any annotation. The annotation \mathbf{c} means “compile-time” and indicates erasable portions of a program.

All Π s, λ s, and $@$ s are annotated. Annotations on Π s distinguish between computational dependence (Π^r) and polymorphism (Π^c). In concrete syntax, we would simply write Π for Π^r and \forall for Π^c , but this choice of abstract syntax affords us economy of presentation. These annotations guide the erasure operation to be defined in Section 5.

Type System. Figure 2 contains typing rules for EPTS. There are two forms of judgment, $\Gamma \vdash M :^c A$ and $\Gamma \vdash M :^r A$. The judgment $\Gamma \vdash M :^r A$ says that M is a well-formed run-time entity, while $\Gamma \vdash M :^c A$ says that M is a well-formed compile-time (erasable) entity.

The type system needs to check that all λ s and $@$ s marked \mathbf{c} are erasable. Recalling from Section 1.1 that erasability of λ s and $@$ s in the $\lambda/@$ graph must be considered one connected component (CC) at a time. The flow analysis implicit in the typing rules ensures that every λ and $@$ in the same CC are annotated with the same τ . Therefore, if every λ^c -binder is erasable, then so is every $@^c$ -argument. So we need only verify that each λ^c is erasable — for each $\lambda^c x:A. M$ in the program, all free occurrences of x in M must appear either inside a type annotation or inside an $@^c$ argument.

The typing rules enforce this invariant using the following technique, due to Pfenning [17]. Each λ^c -bound variable x is marked with \mathbf{c} when it is added to the typing context. This mark is then locally switched off (reset) whenever we check a type annotation or $@^c$ argument. We then require that the mark \mathbf{c} has been switched off by the time we reach any occurrence of x . For economy of presentation, an “off” mark in the typing context is represented as an \mathbf{r} mark. Passing this mark/reset/check test guarantees that each λ^c is actually erasable.

Definition (Context Reset Operation) $\boxed{\Gamma^\circ}$

$$\varepsilon^\circ = \varepsilon \qquad (\Gamma, x:\tau A)^\circ = \Gamma^\circ, x:\mathbf{r}A$$

The key steps of the mark/reset/check test are found in the typing rules Π -INTRO (mark), Π -ELIM and RESET (reset), and VAR (check). In particular, notice how rule Π -INTRO marks context entries and Π -ELIM checks function arguments for both $\tau = \mathbf{r}$ and $\tau = \mathbf{c}$.

Rules VAR, WEAK, Π -INTRO, and CONV each have a premise of the form $\Gamma \vdash A :^c s$. The purpose of these rules is to check that A is well-formed *as a type*. Because these rules deal explicitly with types, they use the compile-time typing judgment. In particular, domain annotations are considered as compile-time entities in the Π -INTRO rule.

The Π -FORM rule may seem counter-intuitive at first. Because Π is a type former, one might expect this rule to use \mathbf{c} -judgments rather than \mathbf{r} ones. However, in a dependently typed language, terms may compute (at run-time) to types, so the \mathbf{r} is appropriate. Another possible surprise is that x is marked with \mathbf{r} rather than τ in the typing context of B . This is because the binding site of

the x will never be erased: The only purpose of the context mark c is to enforce erasability of a λ^c .

If we ignore erasure annotations, these typing rules are exactly those of PTS. The extra restrictions on erasure annotations ensure that evaluation of any well typed term never depends on its compile-time portions. We formalize and prove this in Section 5.

Semantics. The default operational semantics of EPTS is simply β -reduction. We do not commit to any particular evaluation order, so the single-step reduction relation is non-deterministic.

Actually this is only one of *two* different operational semantics for EPTS. The remainder of this paper introduces an erasure semantics with potential for more efficient execution.

Meta-theory. The top half of Figure 4 depicts the meta-theory of EPTS. Each box in that figure contains a particular result of the meta-theory. As the development follows closely that of Pure Type Systems, we focus on the changes due to introducing erasure annotations.

First we investigate properties of the context reset operation Γ° . It is idempotent (Lemma 1) and weakens the strength of the typing assumptions (Lemma 2). An admissible phase-weakening rule (Corollary 3) follows immediately from Lemma 2. *Proofs:* Lemma 1 is easily proved by induction on Γ . Lemma 2 is proved by structural induction on the typing derivation. The interesting cases are RESET, where Lemma 1 is used, and VAR and WEAK, which case split on whether Δ is empty. Corollary 3 is an immediate consequence of Lemma 2.

Next, we prove the Substitution Lemma (4). Note that the mode τ_1 of the typing judgment for the term N to be substituted must match the context entry mark of the variable x for which it will be substituted. *Proof:* By induction on the typing derivation. The interesting cases are RESET (requiring Corollary 3) and VAR and WEAK (each proceeding by cases of whether $\Delta = \varepsilon$ or not).

The Coherence Lemma (5) says that our type system is internally coherent in the following way — If we can derive M has type A , then we can also derive that A is a type. *Proof:* By structural induction on the typing derivation. The interesting cases are RESET, using Lemma 1, and II -ELIM, which makes use of Corollary 3 and Lemma 4.

Finally, Subject Reduction (Lemma 6) tells us that evaluation preserves types. Note that the mode τ of the typing judgment is preserved as well as the type. *Proof:* By structural induction on the typing derivation. The most interesting case is II -ELIM in which we use Lemma 4.

4 Implicit Pure Type Systems

IPTS is an implicitly typed (Curry-style) calculus with both explicit and implicit dependent products. This calculus is the target of the erasure operation. This

$$\boxed{\Gamma \vdash M :^{\tau} A}$$

$$\begin{array}{c}
\text{AXIOM} \\
\frac{(s_1, s_2) \in \mathcal{A}}{\vdash s_1 :^{\tau} s_2} \\
\\
\text{VAR} \\
\frac{\Gamma \vdash A :^{\mathcal{C}} s}{\Gamma, x :^{\tau} A \vdash x :^{\tau} A} \\
\\
\text{WEAK} \\
\frac{\Gamma \vdash A :^{\mathcal{C}} s \quad \Gamma \vdash M :^{\tau} B}{\Gamma, x :^{\tau} A \vdash M :^{\tau} B} \\
\\
\text{II-FORM} \\
\frac{(s_1, s_2, s_3) \in \mathcal{R} \quad \Gamma \vdash A :^{\tau} s_1 \quad \Gamma, x :^{\tau} A \vdash B :^{\tau} s_2}{\Gamma \vdash \Pi^{\tau} x : A . B :^{\tau} s_3} \\
\\
\text{II-INTRO} \\
\frac{\Gamma \vdash \Pi^{\tau} x : A . B :^{\mathcal{C}} s \quad \Gamma, x :^{\tau} A \vdash M :^{\tau} B}{\Gamma \vdash \lambda^{\tau} x : A . M :^{\tau} \Pi^{\tau} x : A . B} \\
\\
\text{II-ELIM} \\
\frac{\Gamma \vdash M :^{\tau} \Pi^{\tau} x : A . B \quad \Gamma \vdash N :^{\tau} A}{\Gamma \vdash M @^{\tau} N :^{\tau} B[N/x]} \\
\\
\text{CONV} \\
\frac{\Gamma \vdash M :^{\tau} A \quad \Gamma \vdash B :^{\mathcal{C}} s \quad A =_{\beta} B}{\Gamma \vdash M :^{\tau} B} \\
\\
\text{RESET} \\
\frac{\Gamma^{\circ} \vdash M :^{\tau} A}{\Gamma \vdash M :^{\mathcal{C}} A}
\end{array}$$

Fig. 2. Typing rules for EPTS. (The typing rules for PTS may be obtained from these by ignoring all erasure annotations and removing the then useless RESET rule).

$$\boxed{\Gamma \vdash M : A}$$

$$\begin{array}{c}
\text{AXIOM} \\
\frac{(s_1, s_2) \in \mathcal{A}}{\vdash s_1 : s_2} \\
\\
\text{VAR} \\
\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \\
\\
\text{WEAK} \\
\frac{\Gamma \vdash A : s \quad \Gamma \vdash M : B}{\Gamma, x : A \vdash M : B} \\
\\
\text{II-FORM} \\
\frac{(s_1, s_2, s_3) \in \mathcal{R} \quad \Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A . B : s_3} \\
\\
\text{II-INTRO} \\
\frac{\Gamma \vdash \Pi x : A . B : s \quad \Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x . M : \Pi x : A . B} \\
\\
\text{II-ELIM} \\
\frac{\Gamma \vdash M : \Pi x : A . B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : B[N/x]} \\
\\
\text{III-FORM} \\
\frac{(s_1, s_2, s_3) \in \mathcal{R} \quad \Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \forall x : A . B : s_3} \\
\\
\text{III-INTRO} \\
\frac{x \notin FV(M) \quad \Gamma \vdash \forall x : A . B : s \quad \Gamma, x : A \vdash M : B}{\Gamma \vdash M : \forall x : A . B} \\
\\
\text{III-ELIM} \\
\frac{\Gamma \vdash M : \forall x : A . B \quad \Gamma \vdash N : A}{\Gamma \vdash M : B[N/x]} \\
\\
\text{CONV} \\
\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s \quad A =_{\beta} B}{\Gamma \vdash M : B}
\end{array}$$

Fig. 3. Typing rules for IPTS. Note that \forall -INTRO and \forall -ELIM are not syntax-directed.

calculus is modeled after Miquel’s Implicit Calculus of Constructions (ICC) [14, 13], a Curry-style variant of Luo’s Extended Calculus of Constructions [10]. ICC has a rich notion of subtype that nicely identifies Church encodings with various levels of type refinement.

IPTS is both more and less general than ICC. It is more general because IPTS is defined in terms of an arbitrary PTS specification. It is less general because ICC (1) uses $\beta\eta$ -conversion instead of β -conversion in determining type equality, (2) supports a notion of universe subtyping called *cumulativity* as in Luo’s Extended Calculus of Constructions [10], and (3) contains extra typing rules ensuring η subject reduction and strengthening.

The syntax of IPTS is as follows:

$$\begin{array}{l} \text{(terms)} \quad M, N, A, B ::= x \mid \lambda x. M \mid M N \mid \Pi x:A. B \mid \forall x:A. B \mid s \\ \text{(contexts)} \quad \Gamma, \Delta ::= \varepsilon \mid \Gamma, x:A \end{array}$$

Note the distinction between $\Pi x:A. B$ and $\forall x:A. B$ as well as the omission of domain labels from λ -abstractions.

The difference between explicit and implicit products shows up in the type system (Figure 3). Whereas the explicit product is introduced by functional abstraction (rule Π -INTRO) and eliminated by function application (rule Π -ELIM), no syntactic cues indicate introduction or elimination of the implicit product (rules \forall -INTRO and \forall -ELIM). So Π indicates functional abstraction and \forall indicates *polymorphism*.

5 Erasure

We now define erasure as a translation from EPTS to IPTS.

Definition (Erasure). $\boxed{\Gamma^\bullet}$ and $\boxed{M^\bullet}$

$$\begin{array}{l} \varepsilon^\bullet = \varepsilon \quad (\Gamma, x:\tau A)^\bullet = \Gamma^\bullet, x:A^\bullet \quad x^\bullet = x \quad s^\bullet = s \\ (\Pi^r x:A. B)^\bullet = \Pi x:A^\bullet. B^\bullet \quad (\lambda^r x:A. M)^\bullet = \lambda x. M^\bullet \quad (M@^r N)^\bullet = M^\bullet N^\bullet \\ (\Pi^c x:A. B)^\bullet = \forall x:A^\bullet. B^\bullet \quad (\lambda^c x:A. M)^\bullet = M^\bullet \quad (M@^c N)^\bullet = M^\bullet \end{array}$$

The bottom half of Figure 4 sketches out the meta-theory of erasure. We now discuss the significance of the results listed there.

A pair of key lemmas (7 and 8) characterize which variables occurrences in a term may survive erasure: they are all tagged with r in the typing context. *Proofs:* Lemmas 7 and 8 must be proved simultaneously by structural induction on typing derivations. The WEAK case of the proof of Lemma 7 requires Lemma 8 and the Π -INTRO case of the proof of Lemma 8 requires Lemma 7.

Preservation of Reductions. Since computation happens by substitution, we first show that erasure commutes with substitution (Lemma 9). We then

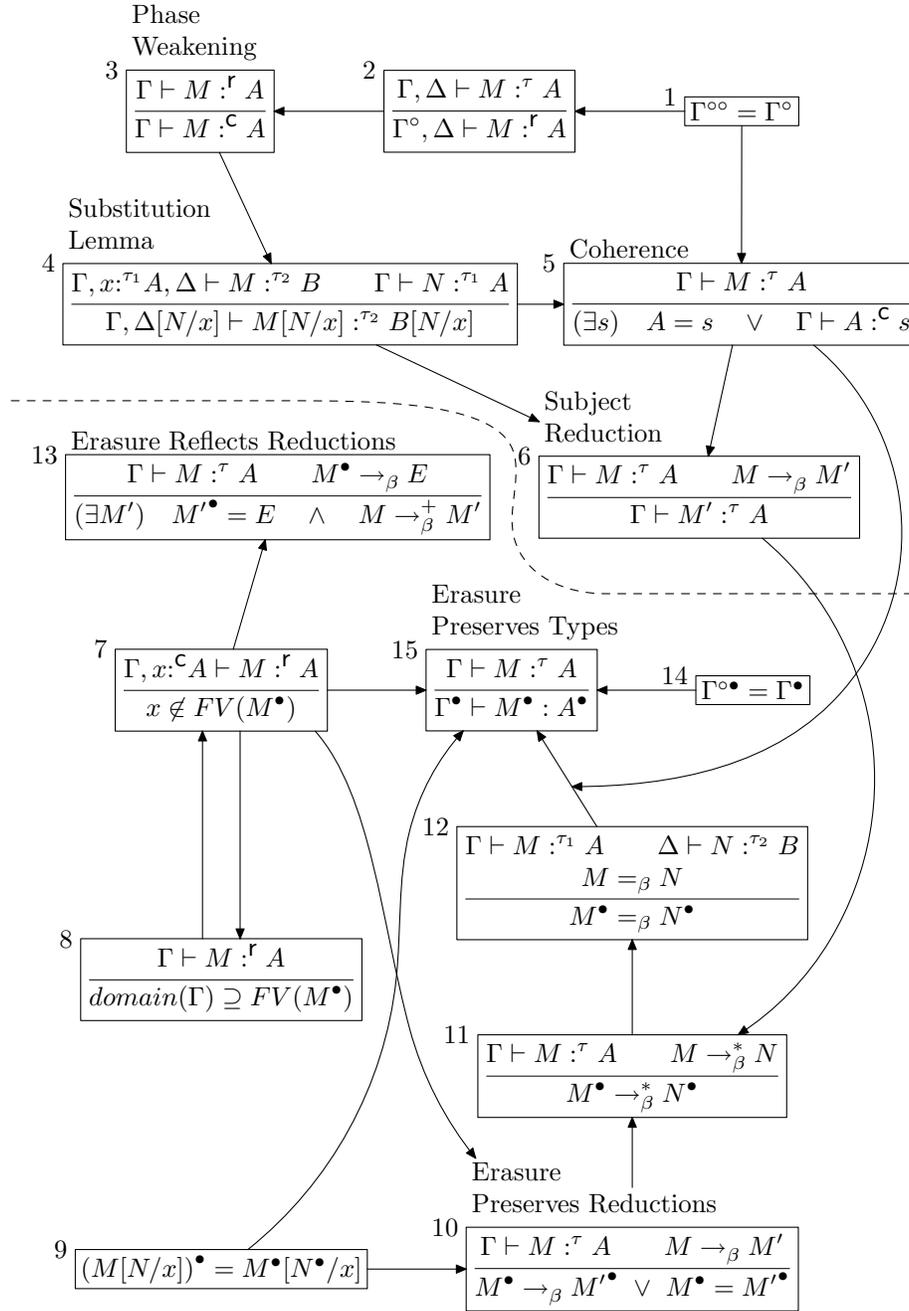


Fig. 4. Identities and admissible rules in the meta-theory of EPTS (above the dotted line) and erasure (below it). Arrows indicate proof dependencies.

show that erasure respects reduction in the following sense: Each reduction step of a well-formed term in EPTS maps to either one *or zero* reduction steps in IPTS (Theorem 10). *Proof:* Lemma 9 is proved by straightforward induction on M . Theorem 10 is by straightforward induction over the typing derivation. The interesting cases are Π -INTRO and Π -ELIM, where we split by cases on τ . In the Π -ELIM case when the reduction step is β , the proof depends on Lemma 9 in the case where $\tau = r$ and on Lemma 7 in the case where $\tau = c$.

The proof of Theorem 10 shows that some EPTS reductions in fact do not work when viewed through the lens of erasure. This is precisely why we want an erasure semantics — to eliminate the work associated with run-time-irrelevant portions of a program. Examination of the proof shows *where* erasure eliminates work. As expected, the eliminated work includes erased redices (terms of the form $(\lambda^c x:A. M)@^c N$, which erase to just M^\bullet) as well as unnecessary reduction steps inside domain-annotations and erased arguments.

Corollaries 11 and 12 follow immediately from Theorem 10. The proof of Corollary 12 also requires the Church-Rosser theorem.

Preservation of Typing. Again, we first investigate the properties of the context reset operation. The erasure operation annihilates it (Lemma 14). *Proof:* By induction on Γ .

Then we prove that erasure preserves well-typedness (Theorem 15). *Proof:* We prove Theorem 15 by structural induction on the typing derivation. The interesting cases are: RESET, in which Lemma 14 is used to simplify $\Gamma^{\circ\bullet}$; Π -INTRO, in which Lemma 7 is used to ensure the premise $x \notin FV(M^\bullet)$ of the \forall -INTRO rule of IPTS; Π -ELIM, in which Lemma 9 is used to simplify the type of the application; and CONV, in which Coherence and Corollary 12 are used to establish the premise $A^\bullet =_\beta B^\bullet$ of the IPTS CONV rule.

Reflection of Reductions. Next we show that a reduction of a post-erasure IPTS term can be reflected back into one *or more* EPTS reductions (Theorem 13). *Proof:* By structural induction on the typing derivation. The interesting case is Π -ELIM when $\tau = r$ and the reduction is a β -step $(\lambda x. P^\bullet) N_0^\bullet \rightarrow_\beta P^\bullet[N_0^\bullet/x]$. In this case, $M = M_0@^r N_0$ and $M_0^\bullet = \lambda x. P^\bullet$ and $E = P^\bullet[N_0^\bullet/x]$. The only way M^\bullet can be $\lambda x. P^\bullet$ is if M_0 is a $\lambda^r x:B. P$ nested under some (perhaps zero) “frames” of the form $\lambda^c y:C. []$ or $[]@^c N$. Because the type of M_0 is $\Pi^r x:A. B$, we know the top-most frame cannot be a λ^c . Similarly, for typing reasons, the bottom-most frame cannot be a $@^c$, because it would be applied to a λ^r . Therefore, if there are any frames at all on top of $\lambda^r x:B. P$, then there are two, and at some point there is a λ^c frame just underneath a $@^c$ one, forming a redex. If we reduce this redex, the rest of the frame structure remains in tact. We repeat this process until no intermediate frames are left. Then $M_0 \rightarrow_\beta^* \lambda^r x:B[\theta]. P[\theta]$ where θ is the simultaneous substitution effected by the sequence of reductions. Because θ is comprised solely of substitutions for λ^c -bound variables, Lemma 7 tells us there will be no occurrences of these

variables inside P^\bullet . Therefore $P[\theta]^\bullet = P^\bullet[\theta^\bullet] = P^\bullet$. Let $M' = P[\theta][N_0/x]$. Then

$$M'^\bullet = P[\theta][N_0/x]^\bullet = P[\theta]^\bullet[N_0^\bullet/x] = P^\bullet[N_0^\bullet/x] = E$$

and $M \rightarrow_\beta^+ M'$ because

$$M = M_0 @^r N_0 \rightarrow_\beta^* (\lambda^r x : B[\theta]. P[\theta]) @^r N_0 \rightarrow_\beta P[\theta][N_0/x] = M',$$

so the proof is finished. \square

The proof of Theorem 13 shows that certain reduction steps in IPTS (of post-erasure EPTS terms) require several additional reductions in the original EPTS term before the reduction corresponding to that in IPTS can take place in EPTS. This means that some of the work that erasure avoids is unavoidable, in general, without erasure.

Theorem 13 says that any post-erasure reduction corresponds to a potential pre-erasure reduction. In other words, the erasure of a well-formed EPTS term cannot reduce in IPTS in a strange way that was not possible in EPTS.

6 Erasure Semantics

The erasure semantics for EPTS is simply this: First erase and then execute in IPTS. The meta-theory supports the claim that this is a good erasure semantics.

Theorem 10 : erasure eliminates some old work

Theorem 13 : erasure does not introduce any new work

Theorem 15 : erasure preserves the meanings (types) of programs

One final result supports the validity of our erasure semantics for EPTS. We would not want a PTS program to compute to a value while some annotation of it diverges under the erasure semantics. Thankfully, this cannot happen.

Theorem (Erasure Preserves Strong Normalization)

For a strongly normalizing PTS, any well-typed term in the corresponding EPTS erases to a strongly normalizing IPTS term.

Proof: Suppose there is an infinite reduction sequence in IPTS starting with the erasure of a well-typed term M in EPTS. By Lemma 6, this reflects back onto an infinite reduction sequence in EPTS starting with M . Because \flat (the erasure-annotation-forgetting map from EPTS to PTS) preserves both reduction steps and typing judgments, we obtain an infinite reduction sequence in the underlying PTS starting with the well-typed term M^\flat . But this contradicts our assumption that the underlying PTS is strong-normalizing. \square

7 Implementation

It should be easy to extend an existing type-checker to handle \forall -types. One must add τ annotations to the abstract syntax and some extra logic to the type-checker to handle these annotations properly. The only potential increase in the time complexity of type-checking comes from the context reset operation.

However, a clever representation of typing contexts renders reset a constant-time operation. The new representation of typing contexts is as follows:

$$\Gamma ::= (\hat{\Gamma}; i) \qquad \hat{\Gamma} ::= \hat{\varepsilon} \mid \hat{\Gamma}, x{:}^i A$$

(The metavariable i denotes an integer). The context operations then become

$$\varepsilon = (\hat{\varepsilon}; 0) \qquad \begin{array}{l} (\hat{\Gamma}; i), x{:}^c A = (\hat{\Gamma}, x{:}^{i+1} A; i) \\ (\hat{\Gamma}; i), x{:}^r A = (\hat{\Gamma}, x{:}^i A; i) \end{array} \qquad (\hat{\Gamma}; i)^\circ = (\hat{\Gamma}; i + 1)$$

$$x{:}^r A \in (\hat{\Gamma}; i) \text{ iff } x{:}^j A \in \hat{\Gamma} \text{ and } j \leq i$$

The top-level i in $\Gamma = (\hat{\Gamma}; i)$ counts how many times prefixes of Γ have been reset. For any binding $x{:}^j A \in \hat{\Gamma}$ originally introduced with the mark τ , we have $j > i$ iff $\tau = c$ and there have been no resets since x was introduced — exactly the condition in which the binding for x would be marked with c in the original implementation.

8 Future Work

8.1 Proof Irrelevance

In a dependently typed language, the conversion typing rule reflects the semantics of a language back into its type system. In EPTS, however, there are two notions of operational semantics. The CONV rule of EPTS reflects the default semantics rather than the erasure semantics. We may attempt to remedy this by modifying the CONV rule as follows:

$$\frac{\text{CONV}^\bullet \quad \Gamma \vdash M :^r A \quad \Gamma \vdash B :^c s \quad A^\bullet =_\beta B^\bullet}{\Gamma \vdash M :^r B}$$

This variant of EPTS is interesting because the CONV[•] rule seems to yield a generalized form of *irrelevance*, including *proof irrelevance* as a special case. Proof irrelevance in a conversion rule means that two proofs are considered equal if they prove the same proposition, regardless of how they each prove it. The CONV[•] rule only requires the run-time portions of A and B to be equal — compile-time portions of A and B (including proofs and perhaps other terms) are considered irrelevant.

Pfenning’s modal variant of LF with built-in notions of intensional code and proof irrelevance [17] provided inspiration for EPTS. The conversion rule of that system seems to us quite similar to CONV[•].

8.2 Parametricity

Languages such as Haskell and ML make heavy use of parametric polymorphism centered around a \forall type constructor. *Parametricity* is a property of such languages enabling one to derive “free theorems” about polymorphic terms based

solely on their type [21]. We conjecture that the \forall -types of EPTS satisfy parametricity properties similar to those of System F.

Many studies of parametricity for System F are based on denotational semantics. It seems impossible to develop a semantic model for an arbitrary EPTS. We think a proof-theoretic approach is necessary, somehow generalizing existing work on System F [18, 2, 11].

9 Conclusions

Languages combining dependent types with erasure semantics sometimes require users to maintain more than one copy of a datatype in order to ensure erasure of some of its values but not others. This problem stems from the treatment of erasability as an intrinsic property of data, rather than a property of the way that data is used.

By treating erasability extrinsically — distinguishing functions that don't depend computationally on their arguments from those that do — we overcome the code duplication problem and arrive at a general form of polymorphism over arbitrary sorts of entities (types, proofs, numbers, et-cetera).

This change of perspective leads to a notion of erasure generalizing both type-erasure and proof-erasure (program-extraction). We hope the resulting notion of computational irrelevance similarly generalizes both proof-irrelevance and parametricity-style notions of representation independence.

Acknowledgments Thanks to Andrew Tolmach, Mark Jones, Jim Hook, Tom Harke, Chuan-Kai Lin, Ki-Yung Ahn, Andrew McCreight, Dan Brown, and John McCall for their comments on this work.

References

1. The Coq proof assistant. <http://coq.inria.fr>.
2. M. Abadi, L. Cardelli, and P.-L. Curien. Formal parametric polymorphism. *Theoretical Computer Science*, 121(1-2):9–58, 1993.
3. Lennart Augustsson. Cayenne – a language with dependent types. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 239–250, 1998.
4. Henk P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.
5. Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *14th International Symposium on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer-Verlag, 2006.
6. Edwin Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, University of Durham, 2005.

7. Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 66–77, 2005.
8. Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 42–54, 2006.
9. Chunxiao Lin, Andrew McCreight, Zhong Shao, Yiyun Chen, and Yu Guo. Foundational typed assembly language with certified garbage collection. In *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*, pages 326–338. IEEE Computer Society Press, 2007.
10. Zhaohui Luo. *Computation and reasoning: a type theory for computer science*. Oxford University Press, New York, USA, 1994.
11. Harry G. Mairson. Outline of a proof theory of parametricity. In *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 313–327, 1991.
12. Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.
13. Alexandre Miquel. The implicit calculus of constructions. In *Proceedings of 5th International Conference on Typed Lambda Calculi and Applications*, volume 2044 of *Lecture Notes in Computer Science*, pages 344–359. Springer-Verlag, 2001.
14. Alexandre Miquel. *Le Calcul des Constructions Implicite: Syntaxe et Sémantique*. PhD thesis, Université Paris 7, 2001.
15. George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 106–119, 1997.
16. Simon Peyton-Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, 2006.
17. Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *LICS'01: Proceedings of the 16th Annual Symposium on Logic in Computer Science*, pages 221–230. IEEE Computer Society Press, June 2001.
18. Gordon D. Plotkin and Martín Abadi. A logic for parametric polymorphism. In *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 361–375. Springer-Verlag, 1993.
19. Tim Sheard. Languages of the future. In *Proceedings of the Nineteenth ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 116–119, 2004. OOPSLA Companion Volume.
20. Tim Sheard and Emir Pašalić. Meta-programming with built-in type equality. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM'04)*, pages 106–124, July 2004. Available at <http://cs-www.cs.yale.edu/homes/carsten/lfm04/>.
21. Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, September 1989.
22. Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, 1999.