

Erasability Analysis for Pure Type Systems

Nathan Mishra Linger Tim Sheard

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

Abstract

Erasure Pure Type Systems are an extension to Pure Type Systems with an erasure semantics. Erasure is guided by lightweight annotations that may be supplied by either a programmer or a prior automatic program analysis phase. We present an algorithm for automatic erasure annotation, generating annotations that are optimal in the sense of marking as much of a program for erasure as possible.

We reduce our program analysis problem to a variation of the well-studied boolean satisfiability problem (SAT), in which we want to maximize the number of propositional variables set to true. We give a polynomial time algorithm for this problem in the special case that every clause in the CNF of the input formula contains exactly one negative literal.

Our methodology of dividing program extraction into separate analysis and erasure phases is analogous to the situation in partial evaluation of dividing partial evaluation into separate binding-time analysis and specialization phases.

Keywords Erasure Semantics, Pure Type Systems

1. Introduction

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 [6, 3, 1, 2]. 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 to establish that the properties hold. 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. In a companion paper [4] we introduce Erasure Pure Type Systems, an approach to combining dependent types with erasure semantics that generalizes the notions of parametric polymorphism and type

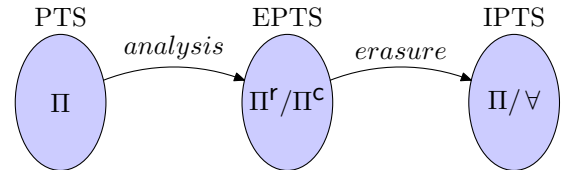


Figure 1. Two-phase approach to erasure semantics for Pure Type Systems in terms of two variants supporting polymorphism. Polymorphic terms are typed explicitly in EPTS and implicitly in IPTS.

erasure in System F to more expressive type theories including dependent types.

That work showed how to extend the highly generic framework of Pure Type Systems with an additional type-former like \forall and corresponding forms of λ -abstraction and application. Just as in System F, we provide an erasure semantics whereby the λ -binders of the new λ -abstractions and the arguments of the new applications may be erased prior to program execution.

The erasure semantics is guided by source language erasure annotations. However, manual program annotation is undesirable or infeasible in some situations (e.g. for large legacy programs). For these reasons, we would like an automatic program analysis to infer erasure annotations. This paper defines such an analysis.

1.1 Methodology

Our study of erasure semantics for dependently typed languages takes place in the context of three different families of typed λ -calculi, shown in Figure 1.

The first family is *Pure Type Systems* (PTS), a well-known formalism that encompasses and organizes a wide variety of type systems. Most dependently typed languages have a PTS at their core, so PTS is a good setting for studying features of dependently typed languages. Appendix A briefly reviews the basics of PTS.

The next two language families are Erasure Pure Type Systems (EPTS) and Implicit Pure Type Systems (IPTS), each of which support a type former indicating polymorphism (Π^c in EPTS and \forall in IPTS). Polymorphic terms in IPTS are implicitly typed, but in EPTS they are explicitly typed and contain erasure annotations marking certain portions of a program for erasure.

Two operations connect the three languages. First a program analysis introduces erasure annotations into a program and then an erasure phase obeys these annotations. Each operation respects the type systems and operational semantics of its source and target languages.

This paper covers the analysis half of Figure 1. A companion paper [4] covers the erasure half. (Appendix B also briefly reviews EPTS and the erasure translation).

The division of labor in Figure 1 is reminiscent of off-line partial evaluation, which consists of two tasks: a binding time

$$\begin{array}{l}
\text{(A) } \text{let } f = \lambda^{\alpha_1} x:\mathbb{N}. 5 \text{ in} \\
\text{let } g = \lambda^{\alpha_2} y:\mathbb{N}. 9 \text{ in} \\
\Rightarrow \text{let } h = \lambda^{\alpha_3} z:(\prod^{\alpha_7} x:\mathbb{N}. \mathbb{N}). z@^{\alpha_4} 7 \text{ in} \\
(h@^{\alpha_5} f, h@^{\alpha_6} g)
\end{array}
\quad
\text{(B) } \left. \begin{array}{l}
(\alpha_7 = \alpha_4) \wedge (\alpha_3 = \alpha_5) \wedge (\alpha_1 = \alpha_7) \wedge (\alpha_3 = \alpha_6) \wedge (\alpha_2 = \alpha_7) \wedge (\neg \alpha_3) \\
(\alpha_7 = \alpha_4) \wedge (\text{false} = \alpha_5) \wedge (\alpha_1 = \alpha_7) \wedge (\text{false} = \alpha_6) \wedge (\alpha_2 = \alpha_7) \quad \leftarrow \\
(\alpha_7 = \alpha_4) \wedge (\alpha_1 = \alpha_7) \wedge (\alpha_2 = \alpha_7) \quad \leftarrow
\end{array} \right\} \text{(C.1)}$$

$$\begin{array}{l}
\text{(C.2) } \alpha_3, \alpha_5, \alpha_6 := \text{false} \\
\alpha_7, \alpha_4, \alpha_1, \alpha_2 := \text{true}
\end{array}
\quad
\text{(D) } \Rightarrow \begin{array}{l}
\text{let } f = \lambda^c x:\mathbb{N}. 5 \text{ in} \\
\text{let } g = \lambda^c y:\mathbb{N}. 9 \text{ in} \\
\text{let } h = \lambda^r z:(\prod^c x:\mathbb{N}. \mathbb{N}). z@^c 7 \text{ in} \\
(h@^r f, h@^r g)
\end{array}
\quad
\text{(E) } \Rightarrow \begin{array}{l}
\text{let } f = 5 \text{ in} \\
\text{let } g = 9 \text{ in} \\
\text{let } h = \lambda z. z \text{ in} \\
(h f, h g)
\end{array}$$

Figure 2. Sketch of erasability analysis and erasure for an example program. Erasability analysis consists of (A) annotation with annotation variables, (B) constraint generation, (C) optimal constraint solution, and (D) solution-determined erasure annotation. Erasure consists of (E) an annotation-guided erasure phase.

analysis phase that annotates a program for specialization and a program specialization phase that simply obeys these annotations. This separation of concerns allows us to see the issues involved more clearly and allows the possibility of programming directly in the annotated language. In partial evaluation, MetaML¹ and its successor MetaOCaml² are each examples of such an annotated intermediate language.

1.2 An example

Figure 2 refines Figure 1 by taking a single example program all the way from through analysis and erasure.

$$\begin{array}{l}
(* \text{ in PTS } *) \\
\text{let } f = \lambda x:\mathbb{N}. 5 \text{ in} \\
\text{let } g = \lambda y:\mathbb{N}. 9 \text{ in} \\
\text{let } h = \lambda z:(\prod x:\mathbb{N}. \mathbb{N}). z 7 \text{ in} \\
(h f, h g)
\end{array}
\quad
\Rightarrow
\quad
\begin{array}{l}
(* \text{ in EPTS } *) \\
\text{let } f = 5 \text{ in} \\
\text{let } g = 9 \text{ in} \\
\text{let } h = \lambda z. z \text{ in} \\
(h f, h g)
\end{array}$$

The program analysis identifies syntactically constant functions and the arguments to which they are applied as candidates for erasure. In a more realistic program, these erasable portions can be quite large. Section 2 further discusses which parts of a program we consider erasable.

First we annotate a program with *annotation variables* that will later be assigned to concrete erasure annotations (A). Then we generate constraints in propositional logic whose solutions correspond to well-formed annotations of the underlying PTS term (B, Section 3). Then we find an optimal solution to the generated constraints corresponding to erasure annotations that mark as much of the program for erasure as possible (C, Section 4). Finally, this optimal solution is applied to the original program, decorating it with concrete erasure annotations (D) that guide the erasure phase (E).

2. Erasability

This section investigates which parts of a program³ may be erased in an erasure-semantics.

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 the λ -abstractions in a program without changing its computational behavior.

Dummy λ -binders. The erasure of domain annotations may cause some λ -binders to become superfluous. Consider the term

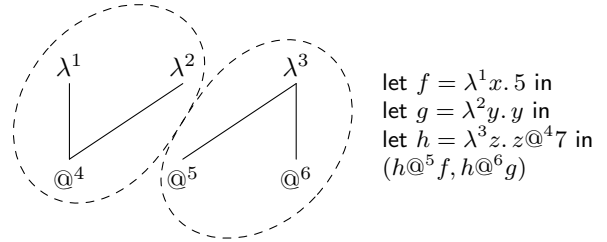


Figure 3. The $\lambda/@$ graph induced by the “may-flow-to” relation of a simple program. The use of y in λ^2 ’s body prevents erasure of both the $@^4$ -argument and the λ^1 -binder.

$(\lambda \alpha:\text{Type}. \lambda x:\alpha. x) \text{Nat } 5$. After erasing type annotations, we are left with $(\lambda \alpha. \lambda x. x) \text{Nat } 5$, 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, during the execution of a program other λ -abstractions may flow to some of those same application sites. We should not erase the argument N at an application site⁴ $M@N$ unless every λ -abstraction that may flow to be the value of M has a dummy binder that ends up getting erased. Similarly, we should not erase a dummy binder unless we also erase every argument to whose application site it may otherwise flow. In general, the “may-flow-to” relation induces a bipartite graph (see Figure 3). In order to decide if a given λ -binder or $@$ -argument may be safely erased, we must analyze all λ -binders in its connected component (CC) in this $\lambda/@$ graph.

Cascading Erasure. Erasure of $@$ -arguments may make other λ -binders into dummies, thereby enabling erasure in other CCs of the

¹ <http://www.cse.ogi.edu/PacSoft/projects/metaml/>

² <http://www.metaocaml.org/>

³ For our purposes here, a program is a term in a typed λ -calculus.

⁴ We sometimes write $@$ for application in order to have a more tangible notation than mere juxtaposition.

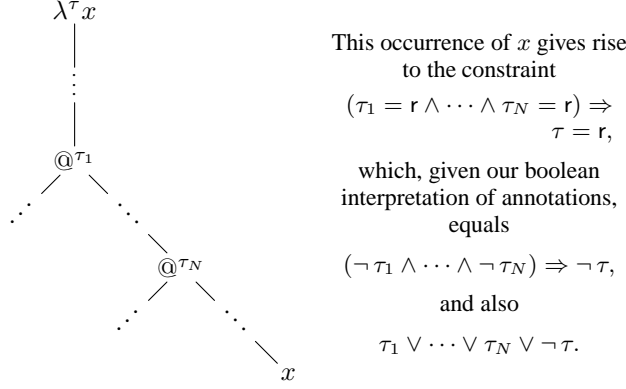


Figure 4. How a typical constraint arises. If each $\tau_i = r$, then this occurrence of x will not be erased, and, therefore, neither can the λ binder. In this case τ must therefore be r .

$\lambda/@$ graph. Consider the following family of identity functions.

```

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

```

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

3. Constraint Generation

In this section, we augment the syntax and typing rules of EPTS (explained in Appendix B) to generate a constraint stating the phase-correctness of a program in terms of its annotation variables. The result is a variant of EPTS called EPTS^C. We then prove that solutions to the generated constraint correspond to legal erasure annotations of the original program.

The generated constraints are formulas of propositional logic with annotation variables doubling as propositional variables. In order to identify erasure annotations and boolean values, we interpret the annotation c as true and r as false. Figure 4 shows how a typical constraint arises.

3.1 Rationale

This section explains the design decisions behind the syntax and type system of EPTS^C.

In order to find annotations for an arbitrary PTS term M , we first annotate each λ , $@$, and Π in M with fresh annotation *variables*. The constraint generation phase then generates a constraint in terms of these annotation variables. Therefore the syntax of EPTS^C terms is as follows:

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

Where the metavariable α denotes an annotation *variable* rather than a concrete annotation $\tau \in \{r, c\}$.

Because applications are annotated with variables, the constraint-generating version of Π -ELIM will have a premise of the form $\Gamma \vdash N :^\alpha A$ where the judgment form is indexed by an annotation variable α . In order to properly handle this case, we need to generalize the RESET rule as follows:

$$\frac{\Gamma^\circ \vdash M :^r A}{\Gamma \vdash M :^c A} \Rightarrow \frac{\Gamma^{\circ(\alpha)} \vdash M :^r A}{\Gamma \vdash M :^\alpha A}$$

where $\Gamma^{\circ(\alpha)}$ is some generalization of Γ° . In order to handle both instantiations of α properly, $\Gamma^{\circ(\alpha)}$ must equal Γ when $\alpha = r$ and Γ° when $\alpha = c$. Therefore we define

$$(\Gamma, x:^\gamma A)^{\circ(\alpha)} = \Gamma^{\circ(\alpha)}, x:^\gamma A$$

where

$$\gamma \circ \alpha = \text{if } \alpha = r \text{ then } \gamma \text{ else } r$$

Under our boolean interpretation of erasure annotations, we transform the definition of $\gamma \circ \alpha$ as follows:

$$\begin{aligned} \gamma \circ \alpha &= \text{if } \neg \alpha \text{ then } \gamma \text{ else false} \\ &= \neg \alpha \wedge \gamma \end{aligned}$$

Each context entry starts out marked with either a r from a Π or a α from a λ . For these reasons, context entry annotations are of the following form:

$$\begin{array}{ll} (\text{context ann.}) & \gamma ::= \alpha \mid r \mid \neg \rho \wedge \gamma \\ (\text{judgment mode}) & \rho ::= \alpha \mid r \mid c \\ (\text{context}) & \Gamma ::= \varepsilon \mid \Gamma, x:^\gamma A \end{array}$$

When typing occurrences of a variable x (in the VAR rule), we require its context annotation γ to be r (false). Therefore, one form of atomic constraint is $\neg \gamma$. The other form of atomic constraint arising from various rules is $\alpha = \alpha'$. The overall constraint is a conjunction of atomic constraints.

$$(\text{constraint}) \quad \mathcal{C}, \mathcal{D}, \mathcal{E} ::= \text{true} \mid \mathcal{C} \wedge \mathcal{D} \mid \neg \gamma \mid \alpha = \alpha'$$

We identify both constraints and context annotations up to logical equivalence.

3.2 Constraint-generating typing rules

Figures 5 and 6 contain the constraint-generating typing rules for our variable-annotated version of EPTS. The judgment forms are $\mathcal{C}; \Gamma \vdash M :^\rho A$ and $\mathcal{C} \vdash M =_\beta N$. The constraint-generating RESET rule makes use of the generalized context reset operation.

DEFINITION 3.1 (Generalized Reset Operation). $\boxed{\Gamma^{\circ(\rho)}}$

$$\varepsilon^{\circ(\rho)} = \varepsilon \quad (\Gamma, x:^\gamma A)^{\circ(\rho)} = \Gamma^{\circ(\rho)}, x:^\gamma A$$

3.3 Proof of correctness

We now prove that the typing rules for EPTS^C are both sound and complete with respect to those of EPTS. In this section, the notation $\sigma \models \mathcal{C}$ means that the variable assignment σ satisfies the formula \mathcal{C} (i.e. \mathcal{C} evaluates to true under σ).

The following four lemmas concern the operation of applying an annotation variable assignment σ to a term or context and how it interacts with other operations and relations such as context reset, substitution, and reduction.

LEMMA 3.2 (Correctness of Generalized Context Reset).

$$\sigma(\Gamma^{\circ(\rho)}) = \begin{cases} \sigma\Gamma & \text{if } \sigma(\rho) = r \\ (\sigma\Gamma)^\circ & \text{if } \sigma(\rho) = c \end{cases}$$

Proof Sketch: By an easy induction on Γ .

LEMMA 3.3. $\sigma(M[N/x]) = \sigma M[\sigma N/x]$

Proof Sketch: Straightforward induction on M .

LEMMA 3.4. If $\sigma P = \widehat{M}[\sigma N/x]$, then $\widehat{M} = \sigma M$ for some M .

Proof Sketch: By straightforward induction on \widehat{M} .

$$\boxed{\mathcal{C}; \Gamma \vdash M :^{\rho} A}$$

$$\begin{array}{c}
\text{AXIOM} \\
\frac{(s_1, s_2) \in \mathcal{A}}{\text{true}; \varepsilon \vdash s_1 :^{\rho} s_2} \\
\\
\text{VAR} \\
\frac{\mathcal{C}; \Gamma \vdash A :^c s}{\mathcal{C} \wedge \neg \gamma; \Gamma, x :^{\gamma} A \vdash x :^{\rho} A} \\
\\
\text{WEAK} \\
\frac{\mathcal{C}; \Gamma \vdash A :^c s \quad \mathcal{D}; \Gamma \vdash M :^{\rho} B}{\mathcal{C} \wedge \mathcal{D}; \Gamma, x :^{\gamma} A \vdash M :^{\rho} B} \\
\\
\text{II-FORM} \\
\frac{\mathcal{C}; \Gamma \vdash A :^{\rho} s_1 \quad \mathcal{D}; \Gamma, x :^{\rho} A \vdash B :^{\rho} s_2}{\mathcal{C} \wedge \mathcal{D}; \Gamma \vdash \Pi^{\alpha} x : A. B :^{\rho} s_3} \quad \frac{\text{II-INTRO}}{\mathcal{C}; \Gamma \vdash \Pi^{\alpha_2} x : A. B :^c s \quad \mathcal{D}; \Gamma, x :^{\alpha_1} A \vdash M :^{\rho} B}{\mathcal{C} \wedge \mathcal{D} \wedge \alpha_1 = \alpha_2; \Gamma \vdash \lambda^{\alpha_1} x : A. M :^{\rho} \Pi^{\alpha_2} x : A. B} \\
\\
\text{II-ELIM} \\
\frac{\mathcal{C}; \Gamma \vdash M :^{\rho} \Pi^{\alpha} x : A. B \quad \mathcal{D}; \Gamma \vdash N :^{\alpha} A}{\mathcal{C} \wedge \mathcal{D}; \Gamma \vdash M @^{\alpha} N :^{\rho} B[N/x]} \\
\\
\text{CONV} \\
\frac{\mathcal{C}; \Gamma \vdash M :^{\rho} A \quad \mathcal{D}; \Gamma \vdash B :^c s \quad \mathcal{E} \vdash A =_{\beta} B}{\mathcal{C} \wedge \mathcal{D} \wedge \mathcal{E}; \Gamma \vdash M :^{\rho} B} \quad \text{RESET} \\
\frac{\mathcal{C}; \Gamma^{\sigma(\rho)} \vdash M :^{\rho} A}{\mathcal{C}; \Gamma \vdash M :^{\rho} A}
\end{array}$$

Figure 5. Constraint generating typing rules for EPTS^C

$$\boxed{\mathcal{C} \vdash M =_{\beta} N}$$

$$\begin{array}{c}
\text{REFL} \\
\frac{}{\text{true} \vdash M =_{\beta} M} \\
\\
\text{SYMM} \\
\frac{\mathcal{C} \vdash M =_{\beta} N}{\mathcal{C} \vdash N =_{\beta} M} \\
\\
\text{TRANS} \\
\frac{\mathcal{C} \vdash M =_{\beta} M'' \quad \mathcal{D} \vdash M'' =_{\beta} M'}{\mathcal{C} \wedge \mathcal{D} \vdash M =_{\beta} M'} \\
\\
\text{BETA} \\
\frac{}{\text{true} \vdash (\lambda^{\alpha} x : A. M) @^{\alpha'} N =_{\beta} M[N/x]} \\
\\
\text{CONGPI} \\
\frac{\mathcal{C} \vdash A =_{\beta} A' \quad \mathcal{D} \vdash B =_{\beta} B'}{\alpha = \alpha' \wedge \mathcal{C} \wedge \mathcal{D} \vdash \Pi^{\alpha} x : A. B =_{\beta} \Pi^{\alpha'} x : A'. B'} \\
\\
\text{CONGLAM} \\
\frac{\mathcal{C} \vdash A =_{\beta} A' \quad \mathcal{D} \vdash M =_{\beta} M'}{\alpha = \alpha' \wedge \mathcal{C} \wedge \mathcal{D} \vdash \lambda^{\alpha} x : A. M =_{\beta} \lambda^{\alpha'} x : A'. M'} \\
\\
\text{CONGAPP} \\
\frac{\mathcal{C} \vdash M =_{\beta} M' \quad \mathcal{D} \vdash N =_{\beta} N'}{\alpha = \alpha' \wedge \mathcal{C} \wedge \mathcal{D} \vdash M @^{\alpha} N =_{\beta} M' @^{\alpha'} N'}
\end{array}$$

Figure 6. Constraint generating conversion rules for EPTS^C

LEMMA 3.5.

$$\frac{\sigma P \rightarrow_{\beta} \hat{Q}}{(\exists Q) \quad \sigma Q = \hat{Q} \quad \wedge \quad P \rightarrow_{\beta} Q}$$

Proof Sketch: By straightforward induction on the structure of the derivation of $\sigma P \rightarrow_{\beta} \hat{Q}$. All the congruence cases are easy. In the case where the reduction step is a single β reduction, the proof makes use of Lemma 3.3

The following two lemmas state that the EPTS^C conversion judgment subsumes the single-step reduction relation and that it is complete for terms with the same underlying structure.

LEMMA 3.6. *If $M \rightarrow_{\beta} N$, then $\text{true} \vdash M =_{\beta} N$.*

Proof Sketch: By induction over the structure of the derivation of $M \rightarrow_{\beta} N$. In the case of a simple β -reduction, use the rule BETA. In any of the congruence cases for the reduction, use the corresponding congruence rule (CONGPI, CONGLAM, or CONGAPP).

LEMMA 3.7 (Pre-Completeness of EPTS^C conversion rules).

$$\frac{\sigma M = \sigma N}{(\exists \mathcal{C}) \quad \mathcal{C} \vdash M =_{\beta} N \quad \wedge \quad \sigma \vDash \mathcal{C}}$$

Proof Sketch: By straightforward induction on σM .

Now we prove soundness and completeness of the EPTS^C conversion rules. The next two theorems say that two EPTS^C terms M and N are provably convertible in EPTS^C under some condition \mathcal{C} satisfied by σ if and only if σ instantiates them to β -convertible terms in EPTS.

THEOREM 3.8 (Soundness of EPTS^C conversion rules).

$$\frac{\mathcal{C} \vdash M =_{\beta} N \quad \sigma \vDash \mathcal{C}}{\sigma M =_{\beta} \sigma N}$$

Proof Sketch: Straightforward induction on the derivation of $\mathcal{C} \vdash M =_{\beta} N$. The interesting cases are BETA, in which we use Lemma 3.3, and the congruence cases, in which we make use of the fact that $\sigma \vDash \alpha = \alpha'$ implies $\sigma \alpha = \sigma \alpha'$. (In fact, the two are logically equivalent).

THEOREM 3.9 (Completeness of EPTS^C conversion rules).

$$\frac{\sigma M =_{\beta} \sigma N}{(\exists \mathcal{C}) \quad \mathcal{C} \vdash M =_{\beta} N \quad \wedge \quad \sigma \vDash \mathcal{C}}$$

Proof: Since $\sigma M =_{\beta} \sigma N$, there exists a term \hat{P} such that $\sigma M \rightarrow_{\beta}^* \hat{P}$ and $\sigma N \rightarrow_{\beta}^* \hat{P}$ (by the Church-Rosser Theorem). By repeated applications of Lemma 3.5, there exists P_1 and P_2 such that $\sigma P_1 =$

$\sigma P_2 = \widehat{P}$ and $M \rightarrow_\beta^* P_1$ and $N \rightarrow_\beta^* P_2$. By Lemma 3.7, there is some constraint \mathcal{C} such that $\mathcal{C} \vdash P_1 =_\beta P_2$ and $\sigma \models \mathcal{C}$. By repeated applications of Lemma 3.6, we have $\text{true} \vdash M =_\beta P_1$ and $\text{true} \vdash N =_\beta P_2$. Therefore, by some applications of SYMM and TRANS, we can derive $\mathcal{C} \vdash M =_\beta N$, and we already know that $\sigma \models \mathcal{C}$. \square

Finally we prove soundness and completeness of the $\text{EPTS}^{\mathcal{C}}$ typing rules. The next two theorems say that an $\text{EPTS}^{\mathcal{C}}$ term M is typable in $\text{EPTS}^{\mathcal{C}}$ under some condition \mathcal{C} satisfied by σ if and only if σ instantiates M to a well-typed EPTS term.

THEOREM 3.10 (Soundness of $\text{EPTS}^{\mathcal{C}}$ typing rules).

$$\frac{\mathcal{C}; \Gamma \vdash M :^{\rho} A \quad \sigma \models \mathcal{C}}{\sigma \Gamma \vdash \sigma M :^{\sigma \rho} \sigma A}$$

Proof Sketch: By straightforward induction on typing derivations. The interesting cases are: VAR, which makes use of our boolean interpretation of formulas; CONV, which makes use of Lemma 3.8; and RESET, which makes use of Lemma 3.2.

THEOREM 3.11 (Completeness of $\text{EPTS}^{\mathcal{C}}$ typing rules).

$$\frac{\sigma \Gamma \vdash \sigma M :^{\sigma \rho} \sigma A}{(\exists \mathcal{C}) \mathcal{C}; \Gamma \vdash M :^{\rho} A \quad \wedge \quad \sigma \models \mathcal{C}}$$

Proof Sketch: By straightforward induction on typing derivations. The interesting cases are: VAR, which makes use of our boolean interpretation of formulas; CONV, which makes use of Lemma 3.8; Π -ELIM, which makes use of Lemma 3.4; and RESET, which makes use of Lemma 3.2.

3.4 Logical structure of generated constraints

Now we investigate the logical structure of context annotations and atomic constraints. Each context annotation γ is a conjunction of a base annotation α or r and the negations of zero or more ρ s. If the base annotation is r (false) then $\gamma = \text{false}$. Similarly, if any ρ is c (true) then $\gamma = \text{false}$. In either case, the atomic constraint $\neg \gamma$ equals true. The remaining case is when the base annotation and each ρ are all variables. Then γ has the form

$$\neg \alpha_1 \wedge \cdots \wedge \neg \alpha_N \wedge \alpha,$$

and, by DeMorgan's laws, the atomic constraint $\neg \gamma$ equals

$$\alpha_1 \vee \cdots \vee \alpha_N \vee \neg \alpha.$$

In other words, non-trivial atomic constraints generated by the VAR rule are logically equivalent to a disjunction of one negated variable and zero or more other variables.

Interestingly, annotation variable equations can also be put in this form:

$$\begin{aligned} \alpha = \alpha' &= (\alpha \Rightarrow \alpha') \wedge (\alpha' \Rightarrow \alpha) \\ &= (\neg \alpha \vee \alpha') \wedge (\neg \alpha' \vee \alpha) \end{aligned}$$

The final conclusion is that the constraints generated by the $\text{EPTS}^{\mathcal{C}}$ typing rules are logically equivalent to a conjunction of disjunctions of one negated variable with zero or more other variables.

3.5 Implementation

Type checking is not decidable for all Pure Type Systems. For strongly normalizing functional Pure Types Systems, there exist algorithmic presentations of the typing rules amenable to direct implementation [8].

We believe a parallel situation holds for Erasure Pure Type Systems. For strongly normalizing functional Pure Type Systems, it should not be too difficult to derive algorithmic versions of the typing rules for the corresponding EPTS that abstractly specify

the behavior of a type-checker. We believe the rules for constraint generation should fit easily into such a type-checker.

4. Constraint Solving

The boolean satisfiability problem (SAT) is to find a satisfying assignment σ for a formula ϕ in propositional logic.

4.1 Terminology

Modern SAT solvers typically take their input formula in *Conjunctive Normal Form* (CNF) — as a conjunction of *clauses* where each clause is a disjunction of *literals*. A literal is either a propositional variable (a *positive* literal) or the negation of a propositional variable (a *negative* literal). The *negation* $\neg L$ of a literal L has the same underlying variable but opposite *sign* (positive or negative). An occurrence of a literal L in a formula is called a *positive occurrence* of L and a *negative occurrence* of $\neg L$. A *unit clause* is a clause consisting of a single literal.

4.2 The TOP-SAT problem

For certain applications some solutions are better than others. We consider the booleans to be totally ordered by setting $\text{true} > \text{false}$. This ordering has a minimum element false and extends pointwise to boolean-valued functions (e.g. variable assignments).

$$\sigma \geq \sigma' \Leftrightarrow \forall X. \sigma(X) \geq \sigma'(X)$$

The *Variable Maximizing SAT Problem* (hereafter TOP-SAT⁵) is as follows: Given a formula ϕ in propositional logic, find a solution σ that is maximal in the pointwise ordering, that is

$$\forall \sigma'. \sigma' \models \phi \Rightarrow \sigma \geq \sigma'.$$

A program solving the TOP-SAT problem for ϕ should first indicate whether a maximal solution for ϕ exists and, if so, give the solution.

In terms of erasure annotations, a maximal solution sets more annotations to c than any other, and therefore marks as much of a program for erasure as possible.

4.3 An algorithm for our special case

The constraints generated by $\text{EPTS}^{\mathcal{C}}$ are in CNF with the special property that each clause contains exactly one negative literal. In this case, there is an efficient algorithm for TOP-SAT.

Let ϕ be a propositional logic formula in CNF with the property that each clause in ϕ contains exactly one negative literal.

$$\phi = (\neg X_1 \vee \varphi_1) \wedge (\neg X_2 \vee \varphi_2) \wedge \cdots \wedge (\neg X_N \vee \varphi_N)$$

(each φ_i is a (possibly empty) disjunction of positive literals). Notice that assigning all variables to false in this situation satisfies ϕ , though this assignment is likely not maximal.

DEFINITION 4.1 (The Algorithm).

1. **Unit Clause Propagation.** While ϕ contains a unit clause, say, L , assign $L = \text{true}$ and simplify ϕ — Remove from ϕ all clauses with positive occurrences of L and remove all negative occurrences of L in other clauses.
2. **Completion.** When no unit clauses are left, assign all remaining unassigned variables to true .

LEMMA 4.2 (Invariant). *Each step of Unit Clause Propagation preserves the invariant that all clauses in ϕ contain exactly one negative literal.*

⁵A more obvious name choice would be “MAX-SAT”, but it already refers to the problem of maximizing the number of satisfied clauses.

Proof: Assuming every clause in ϕ contains a single negative literal, the unit clause that we propagate must consist of a single negative literal $\neg X$. We assign this literal to true (by setting X to false) and simplify. Every clause containing $\neg X$ will be removed and every occurrence of X will be removed from its clause. Each remaining clause still contains its sole negative literal because only positive literals were removed from any (surviving) clause. \square

LEMMA 4.3 (Correctness of Step 1). *If a Unit Clause Propagation step takes ϕ to ϕ' , then any TOP-SAT solution of ϕ' is uniquely extensible to a TOP-SAT solution for ϕ .*

Proof: Because ϕ is a conjunction containing a unit clause $\neg X$, any assignment satisfying ϕ must set X to false. Let σ' be some assignment satisfying ϕ' . Then σ' satisfies every clause in ϕ that was not removed, because each such clause is logically weaker than its corresponding clause in ϕ' . The extended assignment $\sigma'[\text{false}/X]$ also satisfies the clauses that were removed from ϕ . Because σ' maximizes the number of non- X variables set to true in an assignment satisfying ϕ' , so does $\sigma'[\text{false}/X]$ for ϕ , because we may not choose $X = \text{true}$ and still satisfy ϕ . \square

LEMMA 4.4 (Correctness of Step 2). *If ϕ contains no unit clauses and each clause in ϕ contains exactly one negative literal, then $\lambda X. \text{true}$ is the maximal satisfying assignment for ϕ .*

Proof: In this case, ϕ is of the form

$$(\neg X_1 \vee \varphi_1) \wedge (\neg X_2 \vee \varphi_2) \wedge \cdots \wedge (\neg X_N \vee \varphi_N)$$

where each φ_i is a disjunction of positive literals. Because ϕ contains no unit clauses, each φ_i is non-empty. Let σ be the assignment $\lambda X. \text{true}$. Then $\sigma(\varphi_i) = \text{true}$ because φ_i is non-empty and contains only positive literals. Therefore σ satisfies ϕ

$$\begin{aligned} \sigma(\phi) &= \sigma \left(\bigwedge_i \neg X_i \vee \varphi_i \right) = \bigwedge_i \sigma(\neg X_i) \vee \sigma(\varphi_i) \\ &= \bigwedge_i \text{false} \vee \text{true} \\ &= \text{true} \end{aligned}$$

and is clearly the maximum solution. \square

THEOREM 4.5 (Correctness). *If each clause in ϕ contains exactly one negative literal, then this algorithm returns the maximal assignment satisfying ϕ .*

Proof: By Lemma 4.2, each step of Unit Clause Propagation preserving the invariant that each clause contains exactly one negative literal. When the Unit Clause Propagation loop finishes, any remaining clauses are of size ≥ 2 and the invariant holds, so, by Lemma 4.4, setting all as-yet-undetermined variables to true maximally satisfies the remaining formula. By Lemma 4.3, this solution extends to a maximal solution of the original ϕ . \square

Discussion Unit clause propagation can be explained in terms of erasure annotations. Recall from Figure 4 the cause of a typical phase-ordering constraint $\alpha_1 \vee \cdots \vee \alpha_N \vee \neg \alpha$. A unit clause $\neg \alpha$ corresponds to a variable occurrence for which every enclosing α_i in its scope has been determined to equal r , and therefore α must be r . The process is initiated by occurrences of λ -bound variables that do not appear inside any @-arguments (or domain annotations) in their scope (i.e. $N = 0$).

The algorithm deduces in this way which annotations must be r . When no more annotations can be deduced to equal r , we set all remaining variables to c . The algorithm discussed here calculates a sort of greatest fixed-point. Contrast this to the informal least fixed-point algorithm outlined in Section 2.

4.4 Implementation

Modern SAT solvers rely heavily on unit clause propagation and use clever data structures to implement it efficiently. We can use these same techniques to implement our constraint solver.

In a naive implementation of unit propagation, we maintain for each literal L a listing of clauses in which it appears. Any time a literal L is set to false, we visit each clause it appears in to check if that clause has become a unit clause.

The designers of the Chaff SAT solver [5] pioneered a technique called *two watched literals*. Their insight was that we need not visit a clause of original length n to check if it has become unit until it changes from size $n - 2$ to $n - 1$ and this can never happen as long as there are at least two unassigned literals in the clause. This insight leads to an implementation where we pick two unassigned literals in each clause to watch. Then we need visit that clause to check if it has become unit only if one of its two watched literals becomes false (as opposed to whenever any of its literals becomes false). This greatly speeds up unit clause propagation in general.

5. Future Work

One piece missing from the formal development of EPTS^C is a coherence theorem. There may be several ways to prove that a particular term M is well-formed in a particular context Γ . Different derivations will, in general, generate different constraints. If these different constraints have different optimal solutions, then different portions of M will be marked for erasure in each case.

We don't want the (erasure) semantics of a program to depend on the particular way in which it was type-checked. To satisfy ourselves that this cannot happen, we hope to prove something like the following coherence result.

CONJECTURE 5.1 (Coherence).

$$\frac{\begin{array}{l} C_1; \Gamma \vdash M :^{\rho} A_1 \quad \sigma_1 \models C_1 \quad (\forall \sigma'_1. \sigma'_1 \models \phi \Rightarrow \sigma_1 \geq \sigma'_1) \\ C_2; \Gamma \vdash M :^{\rho} A_2 \quad \sigma_2 \models C_2 \quad (\forall \sigma'_2. \sigma'_2 \models \phi \Rightarrow \sigma_2 \geq \sigma'_2) \end{array}}{\sigma_1 \Gamma = \sigma_2 \Gamma \quad \wedge \quad \sigma_1 M = \sigma_2 M}$$

We have not yet fully investigated whether we can prove such a coherence result for EPTS^C, so for now it is left as future work.

Another way around this problem is to give an algorithmic presentation of the EPTS^C typing rules for a particular PTS specification such that there is at most *one* typing derivation for any given term and typing context.

6. Conclusions

We have developed a two-phase constraint generation and solving strategy for determining optimal erasure annotations for PTS terms. The constraint-generation scheme is sound and complete with respect to the EPTS type system that checks (among other things) correctness of erasure annotations. Though our presentation of EPTS^C is not algorithmic, it should be straightforward to adapt to any type-checker for a particular PTS. The constraint solver we describe exploits state of the art data structures and algorithms from modern SAT solvers.

Because the erasure annotations resulting from our approach are provably optimal, programmers need not bother with manual annotation in order to achieve efficient execution of dependently typed programs.

References

- [1] 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.

- [2] 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.
- [3] 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.
- [4] Nathan Mishra Linger and Tim Sheard. Erasure and polymorphism in pure type systems. Available at <http://www.cs.pdx.edu/~rlinger/>, 2007.
- [5] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, pages 530–535. ACM Press, 2001.
- [6] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 106–119, 1997.
- [7] 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.
- [8] L. S. van Benthem Jutting. Typing in pure type systems. *Information and Computation*, 105(1):30–41, 1993.

Appendices

These appendices briefly review Pure Type Systems (Appendix A) and Erasure Pure Type Systems (Appendix B). A companion paper [4] covers EPTS in more detail.

A. Pure Type Systems

The framework of Pure Type Systems organizes type theory by illuminating the common structure in several previously unrelated notions: functional abstraction, parametric polymorphism, type constructors, and dependent types. Pure Type Systems form the functional core of most dependently typed languages, so they provide a good setting for studying new features of such languages.

A.1 Specifications

The definition of a PTS is parameterized by a *specification* consisting of a set \mathcal{S} of *universes* (a.k.a. *sorts*), a set $\mathcal{A} \subseteq \mathcal{S}^2$ of *axioms*, and a set $\mathcal{R} \subseteq \mathcal{S}^3$ of *rules*. The syntax of a PTS depends on \mathcal{S} and the typing rules depend on \mathcal{A} and \mathcal{R} . We assume a fixed specification throughout the development.

A.2 Syntax

The syntax of PTS terms and contexts is as follows:

$$\begin{aligned} M, N, A, B & ::= x \mid \lambda x:A. M \mid M N \mid \Pi x:A. B \mid s \\ \Gamma, \Delta & ::= \varepsilon \mid \Gamma, x:A \end{aligned}$$

The metavariable x indicates an arbitrary variable, and s an arbitrary sort in \mathcal{S} . Note that there are no distinct syntactic categories for types and terms. This is because certain constructions may be replicated at many levels in the type hierarchy with similar typing rules.

The term $\Pi x:A. B$ is a function type with domain A and codomain $B(x)$ where x stands for the value to which the function is ultimately applied. In this way, the return type of a function may depend on the *value* of the function parameter. When $x \notin FV(B)$ the type $\Pi x:A. B$ indicates a regular (non-dependent) function and may be abbreviated as $A \rightarrow B$.

A.3 Type System

The typing rules for a PTS are shown in Figure 7. The \mathcal{A} and \mathcal{R} components of the specification determine the typing relationship between sorts (rule AXIOM) and the permitted forms of dependency in the language (rule Π -FORM). The notation $M[N/x]$ indicates the capture-free substitution of N for x in M . The premise $A =_{\beta} B$ in the CONV rule denotes β -convertibility.

B. Erasure Pure Type Systems

Erasure Pure Type Systems (EPTS) extend Pure Type Systems (PTS) with annotations indicating erasable parts of a program. The EPTS type system checks the erasability of the parts so annotated.

B.1 Syntax

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

$$\begin{aligned} M, N, A, B & ::= x \mid \lambda^{\tau} x:A. M \mid M @^{\tau} N \mid \Pi^{\tau} x:A. B \mid s \\ \Gamma, \Delta & ::= \varepsilon \mid \Gamma, x:^{\tau} A \\ \tau & ::= r \mid c \end{aligned}$$

The metavariable τ ranges over erasure annotations. The annotation r means “run-time”. Syntax with this annotation behaves just as it would in PTS without any annotation. The annotation c means “compile-time” and indicates erasable portions of the 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 in the typing rules.

B.2 Type system

Figure 8 contains typing rules for EPTS. There are two judgment forms, $\Gamma \vdash M :^c A$ 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.

Recalling the discussion of erasability in Section 2, the type system needs to check that every CC marked c is erasable. 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 inside either a type annotation or inside an $@^c$ argument.

The typing rules enforce this invariant using the following technique, due to Pfenning [7]. Each λ^c -bound variable x is marked with c when it is added to the typing context. This mark is then locally “switched off” whenever we check a type annotation or $@^c$ argument. We then require that the mark 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 by an r mark. Passing this mark/reset/check test guarantees that each λ^c is actually erasable.

DEFINITION B.1 (Context Reset). $\boxed{\Gamma^{\circ}}$

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

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

The Π -FORM rule may seem counter-intuitive at first. Because Π is a type former, one might expect this rule to use c -judgments rather than r ones. However, in a dependently typed language, terms may compute (at run-time) to types, so the r is appropriate. Another

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

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

Figure 7. Typing rules for PTS

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

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

Figure 8. Typing rules for EPTS

possible surprise is that x is marked with τ 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 .

Finally note that rules VAR, WEAK, II-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 a well-formed type. Because these rules deal explicitly with *types*, they use the compile-time typing judgment.

B.3 Erasure semantics

The erasure semantics for EPTS consists of an erasure phase translating EPTS terms into IPTS followed by execution in IPTS. The erasure operation is defined as follows:

DEFINITION B.2 (Erasure). $\boxed{M^{\bullet}}$

$$\begin{array}{ll}
x^{\bullet} = x & s^{\bullet} = s \\
(\Pi^{\tau} x:A. B)^{\bullet} = \Pi x:A^{\bullet}. B^{\bullet} & (\Pi^c x:A. B)^{\bullet} = \forall x:A^{\bullet}. B^{\bullet} \\
(\lambda^{\tau} x:A. M)^{\bullet} = \lambda x. M^{\bullet} & (\lambda^c x:A. M)^{\bullet} = M^{\bullet} \\
(M @^{\tau} N)^{\bullet} = M^{\bullet} N^{\bullet} & (M @^c N)^{\bullet} = M^{\bullet}
\end{array}$$

The relationship between pre- and post-erasure reductions is captured in the following two theorems. Erasure both preserves

$$\frac{\Gamma \vdash M :^{\tau} A \quad M \rightarrow_{\beta} M'}{M^{\bullet} \rightarrow_{\beta} M'^{\bullet} \vee M^{\bullet} = M'^{\bullet}}$$

and reflects

$$\frac{\Gamma \vdash M :^{\tau} A \quad M^{\bullet} \rightarrow_{\beta} N}{(\exists M') \quad M'^{\bullet} = N \quad \wedge \quad M \rightarrow_{\beta}^+ M'}$$

evaluation steps. Each pre-erasure reduction corresponds to *at most* one post-erasure reduction and each post-erasure reduction corresponds to *at least* one pre-erasure reduction. The fact that certain pre-erasure steps do not happen post-erasure is the entire point of the erasure phase — getting rid of unnecessary work.

Erasure also respects types.

$$\frac{\Gamma \vdash M :^{\tau} A}{\Gamma^{\bullet} \vdash M^{\bullet} : A^{\bullet}}$$

In other words, a term evaluates to more or less the same type of value after the erasure phase as it would have without undergoing any erasure.