A Dependently Typed Calculus with Pattern Matching and Erasure Inference

MATÚŠ TEJIŠČÁK, University of St Andrews, United Kingdom

Some parts of dependently typed programs constitute evidence of their type-correctness and, once checked, are unnecessary for execution. These parts can easily become asymptotically larger than the remaining runtime-useful computation, which can cause normally linear-time programs run in exponential time, or worse. We should not make programs run slower by just describing them more precisely.

Current dependently typed systems do not erase such computation satisfactorily. By modelling erasure indirectly through type universes or irrelevance, they impose the limitations of these means to erasure. Some useless computation then cannot be erased and idiomatic programs remain asymptotically sub-optimal.

In this paper, we explain why we need erasure, that it is different from other concepts like irrelevance, and propose a dependently typed calculus with pattern matching with erasure annotations to model it. We show that erasure in well-typed programs is sound in that it commutes with reduction. Assuming the Church-Rosser property, erasure furthermore preserves convertibility in general.

We also give an erasure inference algorithm for erasure-unannotated or partially annotated programs and prove it sound, complete, and optimal with respect to the typing rules of the calculus.

Finally, we show that this erasure method is effective in that it can not only recover the expected asymptotic complexity in compiled programs at run time, but it can also shorten compilation times.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Functional languages**; **Automated static analysis**; *Patterns*; • **Theory of computation** → Type theory.

Additional Key Words and Phrases: dependent types, erasure, inference

ACM Reference Format:
Matúš Tejiščák. 2020. A Dependently Typed Calculus with Pattern Matching and Erasure Inference. *Proc. ACM Program. Lang.* 4, ICFP, Article 91 (August 2020), 29 pages. https://doi.org/10.1145/3408973

1 INTRODUCTION

Consider the following fragment of an Idris program that computes the successor of a binary number. It includes a definition of binary numbers, indexed by their value as natural numbers, and the type signature of add1, which guarantees that the result of add1 must indeed be the successor of the given binary number.

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Bin} : \mathbb{N} \to \mathsf{Type}\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathsf{N} & : & \mathsf{Bin}\ 0 \\
\mathsf{I} & : & \mathsf{Bin}\ k \to \mathsf{Bin}\ (1 + 2 * k) \\
\mathsf{O} & : & \mathsf{Bin}\ k \to \mathsf{Bin}\ (0 + 2 * k)
\end{array} \\
\\
\mathsf{add1} : \mathsf{Bin}\ n \to \mathsf{Bin}\ (1 + n)
\end{array}
$$

Author's address: Matúš Tejiščák, School of Computer Science, University of St Andrews, St Andrews, Fife, United Kingdom, ziman@functor.sk.

2475-1421/2020/8-ART91
https://doi.org/10.1145/3408973

If you try to implement this program in a dependently typed system of your choice and run it, you may discover that it needs $O(2^w)$ space (and thus time) to compute the successor of a w-bit number!

To see why, we need to look at the elaborated form of the above, which reveals implicit arguments, originally not seen in the surface syntax.

$$\begin{array}{l}\textbf{data}\ \mathsf{Bin} : \mathbb{N} \to \mathsf{Type}\ \textbf{where} \\ \quad \mathsf{N} \ : \ \mathsf{Bin}\ 0 \\ \quad \mathsf{I} \ : \ \{k : \mathbb{N}\} \to \mathsf{Bin}\ k \to \mathsf{Bin}\ (1 + 2 * k) \\ \quad \mathsf{O} \ : \ \{k : \mathbb{N}\} \to \mathsf{Bin}\ k \to \mathsf{Bin}\ (0 + 2 * k)\end{array}$$

$$\mathsf{add1} : \{n : \mathbb{N}\} \to \mathsf{Bin}\ n \to \mathsf{Bin}\ (1 + n)$$

The constructors I and O contain a field that holds the *unary* representation of about the half of the represented number. We therefore need to store the exponentially sized unary counterparts of our binary numbers. What's worse, in addition to storing these numbers, we need to perform arithmetic on the unary indices, too. This is illustrated using red colour in the (somewhat contrived) function below, which doubles the given binary number and adds one to the result.

$$\begin{array}{l}\mathsf{doublePlusOne} : \{n : \mathbb{N}\} \to \mathsf{Bin}\ n \to \mathsf{Bin}\ (1 + n + n) \\ \mathsf{doublePlusOne}\ \{n\}\ x = \mathsf{add1}\ \{n + n\}\ (\mathsf{binPlus}\ \{n\}\ \{n\}\ x\ x)\end{array}$$

This overhead is thus not just constant. Dependent typing changes the *asymptotic complexity* of the programs we write, which is not acceptable.

While systems like Coq or Agda have traditionally provided their own means to erase some proofs before code extraction, those means cannot erase fields k and n above, which are not proofs; they are *indices* of the type family Bin.

In this paper, we present an automatic technique for discovering and erasing such compile-time-only data in dependently typed programs with pattern matching.

Contributions.

- We explain why erasure of compile-time data is necessary, why current systems do not remove it sufficiently, and that it differs from irrelevance. To our knowledge, this is the first in-depth explanation of the issue in the literature.
- We present a dependently typed calculus with dependent pattern matching and (optional) erasure annotations, and give its typing rules which check types and erasure annotations.
- For this calculus, we prove commutativity of erasure with single-step reduction in well-typed terms. Assuming confluence, we also prove subject reduction, which implies commutativity of erasure with multi-step reduction.
- We give an erasure inference algorithm, which fills in all erasure annotations that have not been provided by the programmer.
- We prove that the above algorithm is sound and complete with respect to the typing rules and optimal in the number of annotations marked as erased.
- We show that the erasure algorithm is effective in practice, and that it can make not only the compiled programs run faster, but it can make also the compiler itself run faster.

2 MOTIVATION

The existing approaches do not work well in cases like the presented one with binary numbers.

2.1 Intrinsic Erasure

In Coq, we might attempt to place the indices in Prop, a special universe of (erasable) proof types.

The direct approach requires copying the type $\mathbb{N}$ in Prop, together with all functions operating on this new type, such as $(+)$ or $(*)$, which causes duplication. It also fails in functions like the run-length compression function for lists, rleCompress : $(xs : \mathsf{List}\ \mathsf{Int}) \to \mathsf{RLE}\ xs$, where $\mathsf{RLE} : \mathsf{List}\ a \to \mathsf{Type}$ is the type of RLE-compressed views of a list, indexed with the original list. The list xs must not be in Prop because rleCompress needs it at runtime. However, then we cannot erase this index anywhere else.

We could attempt compress : $(xs : \mathsf{List}\ \mathsf{Int}) \to \mathsf{RLE}\ (\mathsf{typeToProp}\ xs)$ but this complicates reasoning, especially given that typeToProp cannot be injective [Dockins 2014]. This issue, and others, are further discussed in Section 6.2.11.

Finally, this this approach is not compatible with proof irrelevance.

2.2 Irrelevance

Irrelevance is the property of language elements that makes them ignored in definitional equality. This is stronger than erasability, which talks only about the unnecessity to retain the values at runtime.

Unfortunately, the terminology in the literature is not consistent. Erasability is sometimes called irrelevance [Eisenberg 2016], run-time irrelevance [Agda authors 2020b], or external erasure [Abel 2011]. Irrelevance is sometimes called compile-time irrelevance [Agda authors 2020a] or internal erasure [Abel 2011].

We could make the values that we want to erase irrelevant since irrelevant values can be erased before runtime. Some literature therefore takes the approach of *identifying* erasure with irrelevance [Barras and Bernardo 2008; Sjöberg 2015]; some [Mishra-Linger 2008] also consider this as an optional design choice.

Irrelevant terms can be disregarded in equality already at the stage of type checking, making type checking more efficient. Furthermore, a system with only one notion of "irrelevance and erasure" might be considered simpler than one with a distinction between the two.

However, we should be cautious before choosing to identify the two concepts.

2.2.1 *Irrelevance and Indices.*

In Agda, we might attempt to make the index of Bin irrelevant [Agda authors 2020a]. However, irrelevance is too strong. With $\mathsf{Bin} : .(n : \mathbb{N}) \to \mathsf{Type}$, where the dot marks the index as irrelevant, any value with type $\mathsf{Bin}\ m$ is accepted as a value of $\mathsf{Bin}\ n$ as well, even if $m \neq n$. The value of the index is ignored, which defeats the purpose of having an index in the first place.

2.2.2 *Mobility of Values.*

The above problem arises because if we make the first argument k irrelevant in constructor $\mathsf{O} : \{k : \mathbb{N}\} \to \mathsf{Bin}\ k \to \mathsf{Bin}\ (2 * k)$, then the type of its second argument, $\mathsf{Bin}\ k$, is ill-formed in Agda unless the index of Bin is also irrelevant.

This issue might be partly addressed by a generalised version of Agda's *shape irrelevance* [Abel 2017] (discussed briefly in Section 6.2.5), or fully by Agda's *runtime irrelevance* [Agda authors 2020b], or by using a calculus with untyped equality such as ICC* [Barras and Bernardo 2008] or Zombie [Sjöberg 2015].

2.2.3 *Not All Erased Values Should Be Irrelevant.*

Consider the program in Figure 1. In this program, the argument b of constructor C might or might not be erasable, depending on how it's used elsewhere in the program. However, we do not want to make it *irrelevant* because that would weaken the type signature of f, making f non-total.

```
data T : Type where
  C : (b : Bool) → T

data U : T → Type where
  UT : U (C True)
  UF : U (C False)

f : U (C True) → Bool
f UT = True
```

Fig. 1. A program where irrelevance is not desirable.

2.2.4 Erasure Inference Is Harder with Irrelevance. If erasability implies irrelevance, then erasure influences equality and erasure inference must be interleaved with type checking. This is more complicated than just solving erasure constraints separately after type checking, and also complicates reasoning about such systems.

2.2.5 Existing Literature. There is prior work that does not equate irrelevance and erasure [Atkey 2018; McBride 2016; Miquel 2001; Mishra-Linger 2008].

2.3 Laziness

Laziness per se does not solve the problem, either. We have not introduced notation to talk about this precisely yet but let's sketch the argument informally. The following function vlen calculates the length of a vector.

```
vlen : (n : ℕ) → Vec n a → ℕ
vlen     [Z]          Nil          =  Z
vlen  ([S] k)  (Cons [k] x xs)     =  S (vlen k xs)
```

The square brackets above represent forced knowledge: by matching on the empty vector in the first clause, we know that its length is Z, and thus we needn't match on it to check. In the second clause, we learn that the length index is a successor of some number k. By knowing the number k, we also know the index of the constructor Cons, which we therefore mark as forced, too. (There are other ways to choose forcing in this scenario; this one highlights the difference.)

If the first argument of vlen were just lazy but not erased, the function would have to force it in order to extract the value k for the recursive call. Erasure inference can however (inductively) spot that k is unused in the recursive call. Since we don't need to check the constructor, nor do we need to extract the value k, the whole argument can be erased.

Finally, laziness used this way is a run-time check, dynamic optimisation, and does not remove all overhead (although it can reduce its impact). It would be better to erase at compile time, statically guarantee that certain computations will not take place at run time, and potentially inform other parts of the compiler.

2.4 Forcing, Detagging, and Collapsing

Originally, Idris used erasure via forcing, detagging, and collapsing [Brady et al. 2004]. It uses the fact that whenever we match on a data constructor in a pattern-matching function, we always have the index of the corresponding type constructor available in another argument of the function.

number	i	::=	$0 \mid 1 \mid 2 \mid \ldots$
name	n, c, f	::=	$\mathsf{a} \mid \mathsf{b} \mid \mathsf{c} \mid \ldots$
term	$T, M, N, R, F, X, \sigma, \tau, \rho$	::=	$n \mid \lambda n :_s \sigma.\ T \mid T\ \widehat{r}\ T \mid \Box$
		$\mid$	$\textbf{let}\ d\ \textbf{in}\ T \mid (n :_s \sigma) \to \rho$
erasure annotation	r, s, t, q	::=	$\mathsf{R} \mid \mathsf{E} \mid i \mid \bullet$
context, telescope	Γ, Π	::=	$\diamond \mid \Gamma, d$
name binding	d	::=	$n :_s \sigma = b$
body of name binding	b	::=	$T \mid \overline{C} \mid \textbf{constructor} \mid \textbf{variable}$
pattern clause	C	::=	$\overline{(n :_s \sigma)}.\ P = T$
pattern	P, L	::=	$n \mid P\ \widehat{r}\ P \mid \lfloor T \rfloor \mid \lceil n \rceil \mid \lfloor f \rfloor$

Fig. 2. Syntax of $\mathsf{TT}_\star$

The forcing optimisation observes that if we can always recover the index fields of the data constructor from the index of the type constructor, then thanks to the availability of the indices, we need not store these fields of that data constructor at all.

The detagging optimisation observes that if we can recover the constructor tag from the type index, we need not store the tag.

Finally, the collapsing optimisation observes that if a type family is detaggable and all constructor fields are either forced index fields or recursive fields, values of this type family are always fully determined by its indices and we never need to store them at all.

This can erase some important classes of data, such as accessibility and domain predicates, which are typically entirely collapsible – and it can erase them already at compile time, making compilation more efficient.

However, this method is useful only when constructor arguments can be recovered from their indices cheaply, such as by matching. For example, in the constructor $\mathsf{Cons} : \{a : \mathsf{Type}\} \to \{n : \mathbb{N}\} \to (x : a) \to (xs : \mathsf{Vect}\ n\ a) \to \mathsf{Vect}\ (\mathsf{S}\ n)\ a$, we can effectively recover the argument n from the index $(\mathsf{S}\ n)$.

Unfortunately, with constructor $\mathsf{O} : \{n : \mathbb{N}\} \to \mathsf{Bin}\ n \to \mathsf{Bin}\ (2 * n)$, recovery of n would require inversion of function $(2*)$, and arbitrary user-defined functions need not even be injective.

In other words, forcing, detagging and collapsing represent "lossless erasure": they remove duplicated information, such as indices that are already present somewhere else. However, we sometimes want to erase irreversibly.

3 A DEPENDENTLY TYPED CALCULUS WITH ERASURE

To address these issues, we propose $\mathsf{TT}_\star$, a dependently typed calculus with erasure built into the type system.

3.1 Syntax

Figure 2 shows the syntax of $\mathsf{TT}_\star$, which consists of:

- an infinite supply of numbers and names;
- terms (which includes types): variables, lambdas, (erasure-annotated) applications, placeholder for erased terms ($\Box$), let expressions and dependent products;
- erasure annotations: R stands for "retained", E stands for "erased", numbered erasure variables (*evars*) represent annotations that are not known yet, the symbol $\bullet$ represents the lack of an annotation;

```
let ℕ : Type = constructor in
  let Z : ℕ = constructor in
    let S : (n : ℕ) → ℕ = constructor in
      let pred : (x : ℕ) → ℕ =
                    ⌊pred⌋   Z   = Z
                (n : ℕ). ⌊pred⌋ (S n) = n
      in pred (S Z)
```

```
let ℕ :0 Type = constructor in
  let Z :1 ℕ = constructor in
    let S :2 (n :3 ℕ) → ℕ = constructor in
      let pred :4 (x :5 ℕ) → ℕ =
                    ⌊pred⌋ 6̂    Z    = Z
                (n :7 ℕ). ⌊pred⌋ 8̂ (S 9̂ n) = n
      in pred 10̂ (S 11̂ Z)
```

Fig. 3. An example $\mathsf{TT}_\star$ program computing the predecessor of 1, without and with explicit evars

- contexts, which are sequences of name bindings
- name bindings, which contain the name being bound, its erasure annotation and type, together with its body;
- binding bodies, which are either plain terms, sequences of pattern clauses, or the special values **constructor** or **variable**;
- pattern clauses, which contain a sequence of pattern variable bindings, a pattern on the left hand side and a term on the right hand side;
- patterns: names, (erasure-annotated) applications, forced patterns, forced constructors, and a special syntactic element used on the left hand side of pattern clauses to refer to the name of the whole function.

A $\mathsf{TT}_\star$ program is a single term, where all definitions are let-bound locally, even type and data constructors, as illustrated in the tiny example in Figure 3. The pattern matching facility of $\mathsf{TT}_\star$ are pattern clauses and there are no case expressions. Forced patterns and forced constructors (also known as *presupposed* constructors [Goguen et al. 2006]) are marked explicitly.

The central feature of $\mathsf{TT}_\star$ are erasure annotations on every name binding (on the colon of binding $n :_r \tau$), and between the operator and the operand of every term or pattern application ($F\ \widehat{r}\ X$).

Annotation E says that the name binding or argument of application can be erased before runtime, while annotation R says that they have to be retained. Annotation i represents an unknown erasure annotation, where i is the number of the corresponding erasure variable (*evar*). We identify evars with their numbers. The symbol $\bullet$ stands for an evar that has not been numbered yet.

Notation.

- $n :_s \sigma$ without a body stands for $n :_s \sigma =$ **variable**.
- $n : \sigma$ without an erasure annotation stands for $n :_\bullet \sigma$.
- Binding $n = b$ without a type stands for $n :_\bullet \square = b$.
- $\overline{x}$ stands for "sequence of xs" and we will usually refer to the individual elements of the sequence as x_i.
- $f\,\widehat{\overline{r}}\overline{X}$ stands for $f\ \widehat{r_1}\ X_1\ \widehat{r_2}\ X_2 \ldots \widehat{r_k}\ X_k$. This is a slight abuse of notation; application is, of course, left-associative.
- $\mathrm{BV}(\Gamma)$ stands for the set of names bound in Γ.
- Γ, Π stands for the concatenation of telescopes Γ and Π into a single telescope.
- $\Gamma; \Pi$ is a pair of telescopes Γ and Π. It can be viewed as concatenation while remembering the boundary.
- We assume that all bindings bind distinct names to avoid questions around variable capture.
- We say "retained" and "retention" as the counterpart of "erased" and "erasure".

$$\frac{\Gamma;\Pi \vdash P \parallel_\mu T \qquad \Gamma;\Pi \vdash P' \parallel_\mu T'}{\Gamma;\Pi \vdash P \mathrel{\widehat{r}} P' \parallel_\mu T \mathrel{\widehat{r'}} T'}\ \textsc{MatchApp} \qquad \frac{}{\Gamma;\Pi \vdash \lceil T \rceil \parallel_\mu T'}\ \textsc{MatchForced}$$

$$\frac{(c' :_s \sigma = \textbf{constructor}) \in \Gamma}{\Gamma;\Pi \vdash \lceil c \rceil \parallel_\mu c'}\ \textsc{MatchForcedCtor} \qquad \frac{(f :_s \sigma = \overline{C}) \in \Gamma}{\Gamma;\Pi \vdash \lfloor f \rfloor \parallel_\mu f}\ \textsc{MatchDefName}$$

$$\frac{(c :_s \sigma = \textbf{constructor}) \in \Gamma}{\Gamma;\Pi \vdash c \parallel_\mu c}\ \textsc{MatchCtor} \qquad \frac{(n :_s \sigma = \textbf{variable}) \in \Pi}{\Gamma;\Pi \vdash n \parallel_\mu \mu(n)}\ \textsc{MatchPatVar}$$

Fig. 4. Pattern match
The general form of the judgement is $\Gamma;\Pi \vdash P \parallel_\mu T$ for pattern P, term T, and substitution μ.

$$\frac{\Gamma \vdash X_i \not\parallel X_i'}{c \text{ bound as constructor in } \Gamma \quad \Gamma \vdash c\,\overline{\widehat{r}\,X} \not\parallel c\,\overline{\widehat{r'}\,X'}}\ \textsc{MismatchArg} \qquad \frac{\Gamma \vdash X_i \not\parallel X_i'}{c, c' \text{ constructors in } \Gamma \quad \Gamma \vdash \lceil c \rceil\,\overline{\widehat{r}\,X} \not\parallel c'\,\overline{\widehat{r'}\,X'}}\ \textsc{MismatchArgForced}$$

$$\frac{c \neq c'}{c, c' \text{ constructors in } \Gamma \quad \Gamma \vdash c\,\overline{\widehat{r}\,X} \not\parallel c'\,\overline{\widehat{r'}\,X'}}\ \textsc{MismatchHead} \qquad \frac{\Gamma \vdash X_i \not\parallel X_i' \qquad (f :_s \sigma = \overline{C}) \in \Gamma}{\Gamma \vdash \lfloor f \rfloor \overline{\widehat{r}\,X} \not\parallel f\,\overline{\widehat{r'}\,X'}}\ \textsc{MismatchLHS}$$

Fig. 5. Pattern mismatch
The general form of the judgement is $\Gamma \vdash P \not\parallel T$ for pattern P and term T.
In both rules, the sequences of arguments $\overline{X}$ may be empty.

3.2 Pattern Matching

Before we talk about reduction, we need to define pattern matching and its prerequisites.

A substitution is a function from names to terms. Substitution μ applied to term T is written $T[\mu]$. The domain of a substitution is the set of names changed by the substitution; $\mathrm{Dom}(\mu) = \{n \mid \mu(n) \neq n\}$. Notation $x \mapsto X$ stands for substitution μ such that $\mu(x) = X$ and for all $y \neq x$, $\mu(y) = y$. Hence $T[x \mapsto X]$ substitutes X for x in term T.

The set of free pattern variables of pattern P with pattern variable context Π is given by function $\mathrm{FPV}_\Pi(P)$, where $\mathrm{FPV}_\Pi(F \mathrel{\widehat{r}} X) = \mathrm{FPV}_\Pi(F) \cup \mathrm{FPV}_\Pi(X)$ and $\mathrm{FPV}_\Pi(n) = \{n\}$ if n is bound in Π. For all other pattern forms P, $\mathrm{FPV}_\Pi(P) = \emptyset$.

The judgement $\Gamma;\Pi \vdash P \parallel_\mu T$ says that in context Γ together with context of pattern variables Π, pattern P matches term T, subject to substitution μ. Informally, μ substitutes for exactly the pattern variables in P (names bound in Π), obtaining a term equivalent to T.

The judgement $\Gamma \vdash P \not\parallel T$ says that pattern P can never match term T, no matter what substitution we choose. Context Γ is used to distinguish names of constructors from names of pattern variables.

In a situation where neither match or mismatch holds, we say that matching is *stuck*.

$$\frac{(n :_r \tau = T) \in \Gamma \qquad T \text{ is a term}}{\Gamma \vdash n \rightsquigarrow T}\ \textsc{RedVar} \qquad \frac{}{\Gamma \vdash (\lambda x :_s \sigma.\ T)\ \widehat{r}\ X \rightsquigarrow T[x \mapsto X]}\ \textsc{Redex}$$

$$\frac{n \notin \mathrm{FV}(T)}{\Gamma \vdash (\textbf{let}\ n :_s \sigma = b\ \textbf{in}\ T) \rightsquigarrow T}\ \textsc{RedLetElim} \qquad \frac{}{\begin{array}{l}\Gamma \vdash (\textbf{let}\ d\ \textbf{in}\ F)\ \widehat{r}\ X \\ \qquad \rightsquigarrow \textbf{let}\ d\ \textbf{in}\ (F\ \widehat{r}\ X)\end{array}}\ \textsc{RedLetAppL}$$

$$\frac{\begin{array}{cc}(f :_s \tau = \overline{C}) \in \Gamma & C_i =: (\Pi_i.\ L_i = R_i) \\ \Gamma; \Pi_k \vdash \textsc{PatWF}_f(L_k) & \mathrm{Dom}(\mu) = \mathrm{FPV}_{\Pi_k}(L_k) \\ \forall i < k.\ \left(\Gamma \vdash L_i \not\nmid f\widehat{\overline{r}}\overline{X}\right) & \Gamma; \Pi_k \vdash L_k \parallel_\mu f\widehat{\overline{r}}\overline{X}\end{array}}{\Gamma \vdash f\widehat{\overline{r}}\overline{X} \rightsquigarrow R_k[\mu]}\ \textsc{RedClauses}$$

Fig. 6. Computation rules of $\mathsf{TT}_\star$
The general form of the judgement is $\Gamma \vdash T \rightsquigarrow T'$ for terms T and T'.
Reduction rules of $\mathsf{TT}_\star$ form the structural closure of these computation rules.

3.3 Reduction Rules

Reduction in $\mathsf{TT}_\star$ is the structural closure of the rules shown in Figure 6 and it always takes place in a context. Variables reduce to their definitions, redexes β-reduce, unreferenced let bindings reduce away, let bindings float out of applications.

The most involved reduction rule is RedClauses, the rule that reduces applications of pattern-matching functions. Such an application has the general form $f\widehat{\overline{r}}\overline{X}$, where f is the name of the function, and $\overline{X}$ are its actual parameters, pairwise coupled with erasure annotations $\overline{r}$. The rule says that if the left hand side of the k-th clause matches $f\widehat{\overline{r}}\overline{X}$ then the whole application reduces to $R_k[\mu]$, which is the right hand side of the k-th clause, subject to certain substitution μ.

Rule RedClauses starts with the requirement that f is bound in context Γ as a function defined by pattern clauses $\overline{C}$. For i-th clause C_i, the telescope of its pattern variables is named Π_i, the pattern on the left hand side is named L_i, and the term on the right hand side is named R_i.

The requirement $\Gamma; \Pi \vdash \textsc{PatWF}_f(L_k)$ imposes certain well-formedness requirements on the left hand side of the matching clauses to rule out invalid patterns like $\left(f\ (\mathrm{pred}\ \mathsf{Z})\right)$, where pred is a function, not a constructor. We need this because $\mathsf{TT}_\star$ does not have a syntactic difference between constructors and non-constructors.

The requirement $\forall i < k.\ \left(\Gamma \vdash L_i \not\nmid^f f\widehat{\overline{r}}\overline{X}\right)$ says that before C_k can be attempted, all left hand sides preceding it must mismatch $f\widehat{\overline{r}}\overline{X}$ without getting stuck. Mismatch is defined in Figure 5.

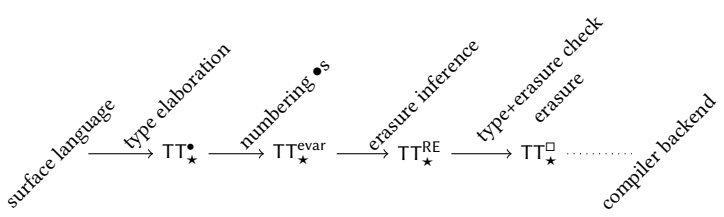
Finally, the requirements $\Gamma; \Pi_k \vdash L_k \parallel_\mu f\widehat{\overline{r}}\overline{X}$ and $\mathrm{Dom}(\mu) = \mathrm{FPV}_{\Pi_k}(L_k)$ say that the left hand side of the k-th clause matches $f\widehat{\overline{r}}\overline{X}$ with substitution μ such that μ substitutes exactly for the free pattern variables in L_k.

With the requirements satisfied, we can reduce $f\widehat{\overline{r}}\overline{X}$ to $R_k[\mu]$, which is the right hand side of the first matching pattern clause modified by the substitution that made its left hand side match.

3.3.1 Conversion. Convertibility is the reflexive symmetric transitive closure of reduction, as shown in Figure 7.

$$\frac{\Gamma \vdash \sigma \rightsquigarrow \tau}{\Gamma \vdash \sigma \approx \tau}\ \textsc{CRed} \qquad \frac{}{\Gamma \vdash \tau \approx \tau}\ \textsc{CRefl} \qquad \frac{\Gamma \vdash \sigma \approx \tau}{\Gamma \vdash \tau \approx \sigma}\ \textsc{CSym} \qquad \frac{\Gamma \vdash \tau \approx \sigma \quad \Gamma \vdash \sigma \approx \rho}{\Gamma \vdash \tau \approx \rho}\ \textsc{CTrans}$$

Fig. 7. Convertibility rules of $\mathsf{TT}_\star$



variant	r ::=	term □
$\mathsf{TT}^{\bullet}_\star$	R, E, •	—
$\mathsf{TT}^{\mathsf{evar}}_\star$	R, E, i	—
$\mathsf{TT}^{\mathsf{RE}}_\star$	R, E	—
$\mathsf{TT}^{\square}_\star$	•	allowed

Fig. 8. Variants of $\mathsf{TT}_\star$

3.3.2 Patterns to Terms. We can turn any pattern P into term $|P|$ using the operation $|\cdot|$ defined as follows.

$$|n| = n \qquad |\lfloor f \rfloor| = f \qquad |\lceil c \rceil| = c \qquad |[T]| = T \qquad |F\ \widehat{r}\ X| = |F|\ \widehat{r}\ |X|$$

3.4 Variants of $\mathsf{TT}_\star$

It is useful to define several variants of $\mathsf{TT}_\star$ for the intermediate steps of the elaboration pipeline, as shown in Figure 8. Each variant is restricted to a different subset of syntax of erasure annotations and may disallow the erased term syntax □. The name $\mathsf{TT}_\star$ stands for any of these variants.

With this distinction, the core calculus is $\mathsf{TT}^{\mathsf{RE}}_\star$, which contains all type and erasure information in a fully explicit form.

3.5 Typing Rules

These rules check the type- and erasure-correctness of $\mathsf{TT}^{\mathsf{RE}}_\star$ terms (Figure 9), bindings (Figure 10), and patterns (Figure 11) in a mutually inductive way. For terms and patterns, the rules are designed to infer the type, if it exists. Rules Conv and PatConv rely on convertibility, which is the reflexive symmetric transitive closure of reduction, as defined in Figure 7.

3.5.1 Lattice of Erasure Annotations. Erasure annotations E and R form the lattice $\mathsf{E} < \mathsf{R}$. This defines relation $\leqslant$ so that $\mathsf{E} \leqslant r$ for any r and $r \leqslant \mathsf{R}$ for any r, and operation $\wedge$ so that $\mathsf{E} \wedge r = \mathsf{E}$ and $\mathsf{R} \wedge r = r$.

3.5.2 Terms. Figure 9 shows the rules for terms. Rule Type shows that $\mathsf{TT}_\star$ uses type-in-type. Universe stratification is orthogonal to the focus of this paper and implementations are expected to bring their own universe stratification.

Rule Ref says that whenever a name is referenced, its binding in the context must be retained at least as much as the reference.

Rule Lam places the abstract variable in the context with the declared erasure annotation s, while McBride [2016] and QTT [Atkey 2018] would place $n :_{s \wedge r} \sigma$ in the context here. We use a different application and substitution rule instead. Similarly, in rule Pi, we use annotation s instead of E on the binding of n.

$$\frac{}{\Gamma \vdash \mathsf{Type} :_r \mathsf{Type}}\ \textsc{Type} \qquad \frac{n :_s \tau = b \ \in \ \Gamma \qquad r \leqslant s}{\Gamma \vdash n :_r \tau}\ \textsc{Ref}$$

$$\frac{\Gamma \vdash \sigma :_{\mathsf{E}} \mathsf{Type} \qquad \Gamma, n :_s \sigma \vdash T :_r \rho}{\Gamma \vdash (\lambda n :_s \sigma.\ T) :_r (n :_s \sigma) \to \rho}\ \textsc{Lam} \qquad \frac{\Gamma \vdash \sigma :_{\mathsf{E}} \mathsf{Type} \qquad \Gamma, n :_s \sigma \vdash \rho :_{\mathsf{E}} \mathsf{Type}}{\Gamma \vdash \big((n :_s \sigma) \to \rho\big) :_r \mathsf{Type}}\ \textsc{Pi}$$

$$\frac{\Gamma \vdash F :_r (n :_s \sigma) \to \rho \qquad \Gamma \vdash X :_{r \wedge s} \sigma}{\Gamma \vdash F \mathrel{\widehat{s}} X :_r \rho[n \mapsto X]}\ \textsc{App}$$

$$\frac{\Gamma \vdash \textsc{BndOk}(d) \qquad \Gamma, d \vdash T :_r \tau}{\Gamma \vdash \big(\textbf{let}\ d\ \textbf{in}\ T\big) :_r \tau}\ \textsc{Let} \qquad \frac{\Gamma \vdash T :_r \tau \qquad \Gamma \vdash \tau \approx \sigma \qquad \Gamma \vdash \sigma :_{\mathsf{E}} \mathsf{Type}}{\Gamma \vdash T :_r \sigma}\ \textsc{Conv}$$

Fig. 9. Erasure checking and type inference rules for $\mathsf{TT}^{\mathsf{RE}}_{\star}$ terms

Rule App illustrates the key point of erasure: the argument X is retained only if the whole term $F \mathrel{\widehat{s}} X$ is retained and if the application itself is retained. Otherwise, X can be erased.

Rule Let checks that the binding is well-typed, places it in the context and then checks the RHS.

Finally, rule Conv embodies the convertibility of types, as defined in Figure 7.

3.5.3 *Forced Patterns.*

Definition 1. We say that the LHS of a clause, $\lfloor f \rfloor_{\widehat{r}} \overline{X}$, is *forced-pattern-consistent* if the following holds for any Γ, Π, $\overline{r'}$, $\overline{X'}$, and τ.

$$\frac{\Gamma; \Pi \vdash \lfloor f \rfloor_{\widehat{r}} \overline{X} \parallel_{\mu} f_{\widehat{r'}} \overline{X'} \qquad \Gamma \vdash f_{\widehat{r'}} \overline{X'} : \tau}{\forall i. \qquad \Gamma \vdash |X_i|[\mu] \approx X'_i}$$

This requirement is also known as *respectfulness of patterns* [Goguen et al. 2006]. In this paper, we do not give an algorithm to check the consistency of forced patterns.

3.5.4 *Bindings.*

Figure 10 shows the rules for bindings. Judgement Bnds checks telescopes of bindings, such as contexts. Judgement Bnd checks individual bindings. Rule BndAbstr checks that the types of the postulated names are well-formed. BndTerm ensures match of the declared and inferred types of the term that defines the binding.

Clause checks the type-correctness of a single clause. It is based on the corresponding rule of TT [Brady 2013], which infers the type of the pattern on the LHS, the type of the term on the RHS, and then checks that they are convertible. This corresponds to the third line of hypotheses of rule Clause.

The first line of hypotheses of Clause checks that the telescope of pattern variable bindings Π is well-typed. Like RedClause, it also checks that the pattern is well-formed and that the free pattern variables on the LHS are exactly the names bound in Π.

The second line of hypotheses of Clause requires that the LHS is linear: contains no repeated pattern variables (pattern variables occurring in forced patterns do not count). It also requires that the LHS be forced-pattern-consistent (as defined in Section 3.5.3).

The general form of the conclusion is $\Gamma \vdash \textsc{Clause}^{f}_{r}(C)$, where C is the clause, f is the name of the function in whose definition the clause occurs, and r is the erasure annotation of the function.

$$\frac{}{\Gamma \vdash \textsc{Bnds}(\diamond)}\ \textsc{BndsBase} \qquad \frac{\Gamma \vdash \textsc{Bnd}(d_1) \qquad \Gamma, d_1 \vdash \textsc{Bnds}(d_2, \ldots, d_n)}{\Gamma \vdash \textsc{Bnds}(d_1, d_2, \ldots, d_n)}\ \textsc{BndsStep}$$

$$\frac{\Gamma \vdash \tau :_{\mathsf{E}} \mathsf{Type} \qquad b \in \{\textbf{variable}, \textbf{constructor}\}}{\Gamma \vdash \textsc{Bnd}(n :_r \tau = b)}\ \textsc{BndAbstr} \qquad \frac{T \text{ is a term} \qquad \Gamma \vdash \tau :_{\mathsf{E}} \mathsf{Type} \qquad \Gamma, n :_r \tau \vdash T :_r \tau}{\Gamma \vdash \textsc{Bnd}(n :_r \tau = T)}\ \textsc{BndTerm}$$

$$\frac{\Gamma \vdash \tau :_{\mathsf{E}} \mathsf{Type} \qquad \forall i.\ \left(\Gamma, n :_r \tau \vdash \textsc{Clause}_r^n(C_i)\right)}{\Gamma \vdash \textsc{Bnd}(n :_r \tau = \overline{C})}\ \textsc{BndClauses}$$

$$\frac{\begin{array}{c}\Gamma \vdash \textsc{Bnds}(\Pi) \qquad \Gamma; \Pi \vdash \textsc{PatWF}_f(\lfloor f \rfloor\, \overline{\widehat{s}P}) \qquad \mathrm{FPV}_\Pi\left(\lfloor f \rfloor\, \overline{\widehat{s}P}\right) = \mathrm{BV}(\Pi) \\ \lfloor f \rfloor\, \overline{\widehat{s}P} \text{ is linear} \qquad \lfloor f \rfloor\, \overline{\widehat{s}P} \text{ is forced-pattern-consistent} \\ \Gamma; \Pi \vdash \lfloor f \rfloor\, \overline{\widehat{s}P} :_r^r \tau \qquad \Gamma, \Pi \vdash R :_r \tau\end{array}}{\Gamma \vdash \textsc{Clause}_r^f\left(\Pi.\ \lfloor f \rfloor\, \overline{\widehat{s}P} = R\right)}\ \textsc{Clause}$$

Fig. 10. Erasure and type checking rules for $\mathsf{TT}^{\mathsf{RE}}_{\star}$ bindings

3.5.5 Patterns. The general form of the pattern typing judgement is $\Gamma; \Pi \vdash P :_r^q \tau$. It contains two contexts, Γ is the usual context and Π is the context of pattern variables declared in the pattern clause. The annotation r says whether the pattern occurs in an erasable place, and the annotation q is the erasure annotation of the whole function being defined.

Rule PatVar states that any pattern variable must be bound with exactly the same retention as the reference to it. Since the flow of information in pattern variables is reversed compared to term variables, we might expect to see the converse of the inequality from the term rule Ref here. However, we require exact equality to make metatheory easier by removing an unnecessary degree of freedom between the declaration of the pattern variable and its occurrence.

Rule PatCtor says that if the whole function is retained (expressed by q), then the pattern inspects and should be retained ($q \leqslant r$) and also the binding of the constructor should be retained ($q \leqslant s$).

Rule PatDefName infers the type for the name of the function being defined. The function is bound in Γ without its body because of BndClauses.

Rule PatForced concatenates contexts Γ and Π into a single context and then checks the forced term in it.

Rules PatApp and PatConv mirror their counterparts for terms and rule PatForcedCtor for constructor $\lceil n \rceil$ essentially mirrors the behaviour of rule PatForced for forced pattern $[n]$.

3.6 Erasure

The erasure operation removes parts of a program not necessary for its execution. Figure 12 shows the definition of the erasure operation $\langle\cdot\rangle$, overloaded for various syntactic classes.

$$\frac{(n :_r \sigma = \textbf{variable}) \in \Pi}{\Gamma; \Pi \vdash n :^q_r \sigma}\ \textsc{PatVar} \qquad \frac{q \leqslant s \qquad q \leqslant r \qquad (n :_s \sigma = \textbf{constructor}) \in \Gamma}{\Gamma; \Pi \vdash n :^q_r \sigma}\ \textsc{PatCtor}$$

$$\frac{(n :_s \sigma = \textbf{constructor}) \in \Gamma \qquad r \leqslant s}{\Gamma; \Pi \vdash \lceil n \rceil :^q_r \sigma}\ \textsc{PatForcedCtor} \qquad \frac{(f :_q \sigma) \in \Gamma}{\Gamma; \Pi \vdash \lfloor f \rfloor :^q_q \sigma}\ \textsc{PatDefName}$$

$$\frac{\Gamma; \Pi \vdash F :^q_r (n :_s \sigma) \to \rho \qquad \Gamma; \Pi \vdash X :^q_{r \wedge s} \sigma}{\Gamma; \Pi \vdash F \mathrel{\widehat{s}} X :^q_r \rho[n \mapsto |X|]}\ \textsc{PatApp} \qquad \frac{\Gamma, \Pi \vdash T :_r \tau}{\Gamma; \Pi \vdash [T] :^q_r \tau}\ \textsc{PatForced}$$

$$\frac{\Gamma; \Pi \vdash P :^q_r \tau \qquad \Gamma, \Pi \vdash \tau \approx \sigma \qquad \Gamma, \Pi \vdash \sigma :_{\mathsf{E}} \mathsf{Type}}{\Gamma; \Pi \vdash P :^q_r \sigma}\ \textsc{PatConv}$$

Fig. 11. Erasure checking, type inference rules for $\mathsf{TT}^{\mathsf{RE}}_{\star}$ patterns

Abstract binding bodies:

$$\begin{array}{lcl} \langle \textbf{constructor} \rangle & = & \textbf{constructor} \\ \langle \textbf{variable} \rangle & = & \textbf{variable} \end{array}$$

Pattern clauses:

$$\left\langle \overline{d}.\ L = R \right\rangle = \left\langle \overline{d} \right\rangle .\ \langle L \rangle = \langle R \rangle$$

Substitutions

$$\langle \mu \rangle\ (n) = \langle \mu(n) \rangle$$

Patterns:

$$\begin{array}{lcl} \langle n \rangle & = & n \\ \left\langle F \mathrel{\widehat{\mathsf{E}}} X \right\rangle & = & \langle F \rangle \\ \left\langle F \mathrel{\widehat{\mathsf{R}}} X \right\rangle & = & \langle F \rangle\ \langle X \rangle \\ \langle [T] \rangle & = & [\langle T \rangle] \\ \langle \lfloor f \rfloor \rangle & = & \lfloor f \rfloor \\ \langle \lceil c \rceil \rangle & = & \lceil c \rceil \end{array}$$

Terms:

$$\begin{array}{lcl} \langle n \rangle & = & n \\ \langle \lambda\, n :_{\mathsf{E}} \tau.\ T \rangle & = & \langle T \rangle \\ \langle \lambda\, n :_{\mathsf{R}} \tau.\ T \rangle & = & \lambda n.\ \langle T \rangle \\ \langle \textbf{let}\ n :_{\mathsf{E}} \tau = b\ \textbf{in}\ T \rangle & = & \langle T \rangle \\ \langle \textbf{let}\ n :_{\mathsf{R}} \tau = b\ \textbf{in}\ T \rangle & = & \textbf{let}\ n = \langle b \rangle\ \textbf{in}\ \langle T \rangle \\ \left\langle F \mathrel{\widehat{\mathsf{E}}} X \right\rangle & = & \langle F \rangle \\ \left\langle F \mathrel{\widehat{\mathsf{R}}} X \right\rangle & = & \langle F \rangle\ \langle X \rangle \\ \langle (n :_r \tau) \to T \rangle & = & \square \end{array}$$

Contexts and telescopes of bindings:

$$\begin{array}{lcl} \langle \diamond \rangle & = & \diamond \\ \left\langle n :_{\mathsf{E}} \tau = b,\ \overline{d} \right\rangle & = & \left\langle \overline{d} \right\rangle \\ \left\langle n :_{\mathsf{R}} \tau = b,\ \overline{d} \right\rangle & = & n = \langle b \rangle\,,\ \left\langle \overline{d} \right\rangle \end{array}$$

Fig. 12. Erasure translation, removing erasable code
These rules use notation shortcuts introduced in Section 3.1.

3.7 Properties

Theorem 1 (Soundness of erasure)**.** *Erasure commutes with single-step reduction in retained terms.*

$$\frac{\Gamma \vdash T :_R \tau \qquad \Gamma \vdash T \rightsquigarrow M}{\begin{array}{c} \textit{either}\ \langle T \rangle = \langle M \rangle \\ \textit{or}\ \langle \Gamma \rangle \vdash \langle T \rangle \rightsquigarrow \langle M \rangle \end{array}}$$

This theorem does not depend on Church-Rosser.

Proof sketch 1. *By induction on the derivation of reduction. The proof mostly relies on basic properties of substitution.*

In the case with pattern matching, we use two lemmas saying that erasure preserves pattern matches, and pattern mismatches. These two lemmas form the bulk of the proof.

Remark 1. As Mishra-Linger notes [2008], this theorem shows why erasure saves work. It happens whenever $\langle T \rangle = \langle M \rangle$ so the erased program need not perform the reduction.

Conjecture 1 (Church-Rosser)**.** *The transitive reflexive closure of $\rightsquigarrow$, written $\rightsquigarrow^\star$, is confluent.*

$$\frac{\Gamma \vdash T \rightsquigarrow^\star M \qquad \Gamma \vdash T \rightsquigarrow^\star M'}{\exists N. \qquad \Gamma \vdash M \rightsquigarrow^\star N \qquad \Gamma \vdash M' \rightsquigarrow^\star N}$$

Remark 2. We have an incomplete proof of this conjecture using parallel reduction. The remaining hole is caused by the need to perform reduction in contexts, arising in the inductive step in **let** expressions. Until we generalise our proof framework to let us express this, we pose Church-Rosser as a conjecture.

Theorem 2 (Subject reduction)**.** *Assuming Church-Rosser, reduction preserves types.*

$$\frac{\Gamma \vdash T :_r \tau \qquad \Gamma \vdash T \rightsquigarrow M}{\Gamma \vdash M :_r \tau}$$

Proof sketch 2. *By induction on the derivation of reduction. The proof mostly relies on basic properties of substitution and the typing relation.*

In the case with pattern matching, we use the Pattern Conversion Lemma. It states that, subject to some additional consistency constraints, if term T matches pattern P with substitution μ, their types are also related through this substitution.

$$\frac{\cdots \textit{additional requirements} \cdots \qquad \Gamma; \Pi \vdash P :^q_r \lambda \qquad \Gamma; \Pi \vdash P \parallel_\mu T \qquad \Gamma \vdash T :_r \tau}{\Gamma \vdash \lambda[\mu] \approx \tau}\ \textsc{PatternConversionLemma}$$

This lemma (and its dependencies) forms the bulk of the proof and establishes the missing link between the unreduced term T and its reduct M, which must have the form $R[\mu]$ for a term R with type λ.

Corollary 1. *Erasure commutes with multi-step reduction in retained terms, assuming Church-Rosser.*

$$\frac{\Gamma \vdash T :_R \tau \qquad \Gamma \vdash T \rightsquigarrow^\star M}{\langle \Gamma \rangle \vdash \langle T \rangle \rightsquigarrow^\star \langle M \rangle}$$

Proof. By iteration of Theorem 1. Use Subject Reduction to carry the well-typedness property along. □

Corollary 2. *Erasure preserves reduction to primitives, such as nullary constructors, integers, Booleans, or strings, assuming Church-Rosser.*

$$\frac{\Gamma \vdash T :_R \tau \qquad \Gamma \vdash T \rightsquigarrow^\star M \qquad \langle M \rangle = M}{\langle \Gamma \rangle \vdash \langle T \rangle \rightsquigarrow^\star M}$$

Proof. Substitute $\langle M \rangle = M$ in Corollary 1. □

$$\frac{\Gamma \vdash \sigma \rightsquigarrow \tau}{\Gamma \vdash \sigma \approx \tau \mid \emptyset}\ \textsc{CRed} \qquad \frac{\tau = \sigma \mid \Delta}{\Gamma \vdash \tau \approx \sigma \mid \Delta}\ \textsc{CRefl} \qquad \frac{\Gamma \vdash \sigma \approx \tau \mid \Delta}{\Gamma \vdash \tau \approx \sigma \mid \Delta}\ \textsc{CSym} \qquad \frac{\Gamma \vdash \tau \approx \sigma \mid \Delta \quad \Gamma \vdash \sigma \approx \rho \mid \Sigma}{\Gamma \vdash \tau \approx \rho \mid \Delta \cup \Sigma}\ \textsc{CTrans}$$

Fig. 13. Conversion rules of $\mathsf{TT}^{\mathsf{evar}}_{\star}$

Corollary 3. *Erasure preserves conversion, assuming Church-Rosser.*

$$\frac{\Gamma \vdash T \approx T' \qquad \Gamma \vdash T :_R \tau \qquad \Gamma \vdash T' :_R \tau'}{\langle \Gamma \rangle \vdash \langle T \rangle \approx \langle T' \rangle}$$

Proof. By Church-Rosser, T and T' have a common reduct. By Corollary 1, this is preserved by erasure. □

4 ERASURE INFERENCE

Erasure inference is elaboration of $\mathsf{TT}^{\bullet}_{\star}$ programs, where some or all erasure annotations are unknown, into $\mathsf{TT}^{\mathsf{RE}}_{\star}$ programs, where each erasure annotation has been assigned a definite value R or E. This happens in the following steps.

(1) Start with a $\mathsf{TT}^{\bullet}_{\star}$ program, which was obtained by erasure-agnostic type elaboration of the program expressed in the surface syntax.
(2) Replace all unknown erasure annotations (•) with unique evars. This translates the program from $\mathsf{TT}^{\bullet}_{\star}$ to $\mathsf{TT}^{\mathsf{evar}}_{\star}$.
(3) Check the program using the constraint-generating typing rules (Section 4.1).
(4) Find a solution of the constraints such that the minimal number of evars is assigned R.
(5) Replace each evar with E or R, according to the solution. This translates the program from $\mathsf{TT}^{\mathsf{evar}}_{\star}$ to $\mathsf{TT}^{\mathsf{RE}}_{\star}$.

Once erasure inference has determined which parts of the program can be erased and annotated them with E, we can erase them, as described in Section 3.6, turning a $\mathsf{TT}^{\mathsf{RE}}_{\star}$ program into a $\mathsf{TT}^{\square}_{\star}$ program, and execute the result more efficiently.

4.1 Constraint-Generating Rules

These rules mostly mirror the type checking rules we saw in Section 3.5. Type-wise, they work the same, always inferring the types of terms and patterns, if they exist.

Compared to the checking rules, here we extend each type-checking or conversion judgement $\mathcal{J}$ with a set of constraints on the right-hand side, written $\mathcal{J} \mid \Delta$, where the constraints in Δ relate the evars occuring in $\mathcal{J}$, and must be satisfied for the judgement to hold. We pronounce this "$\mathcal{J}$, given Δ".

4.1.1 Constraints. Each constraint has the form $G \longrightarrow r$, where G is a set of erasure annotations ("guards") and r is an erasure annotation. Each erasure annotation can be either an evar or a specific value R or E. Intuitively, $G \longrightarrow r$ means that if all annotations in G are retained, r must be retained, too.

We write $r \longrightarrow s$ for $\{\{r\} \longrightarrow s\}$, $r \leftrightarrow s$ for $(r \longrightarrow s) \cup (s \longrightarrow r)$, and $(G \wedge r)$ for $G \cup \{r\}$. We also write $G \longrightarrow X$ for $\{G \longrightarrow r \mid r \in X\}$ and $X \leftrightarrow Y$ for $(X \longrightarrow Y) \cup (Y \longrightarrow X)$.

4.1.2 Conversion. The convertibility judgement generates constraints (Figure 13). Rule CRefl uses constrained equality $\tau = \sigma \mid \Delta$, which says that τ and σ are syntactically equal up to erasure

$$\frac{}{\Gamma \vdash \mathsf{Type} :_G \mathsf{Type} \mid \emptyset}\ \textsc{Axiom} \qquad \frac{(n :_s \tau = b) \in \Gamma}{\Gamma \vdash n :_G \tau \mid G \to s}\ \textsc{Ref}$$

$$\frac{\Gamma \vdash \textsc{Bnd}(n :_s \sigma) \mid \Delta \qquad \Gamma, n :_s \sigma \vdash T :_G \rho \mid \Sigma}{\Gamma \vdash (\lambda\, n :_s \sigma.\, T) :_G (n :_s \sigma) \to \rho \mid \Delta \cup \Sigma}\ \textsc{Lam}$$

$$\frac{\Gamma \vdash \textsc{Bnd}(n :_s \sigma) \mid \Delta \qquad \Gamma, n :_s \sigma \vdash \rho :_{\{\mathsf{E}\}} \mathsf{Type} \mid \Sigma}{\Gamma \vdash \big((n :_s \sigma) \to \rho\big) :_G \mathsf{Type} \mid \Delta \cup \Sigma}\ \textsc{Pi} \qquad \frac{\Gamma \vdash \textsc{Bnd}(d) \mid \Delta \qquad \Gamma, d \vdash T :_G \tau \mid \Sigma}{\Gamma \vdash \big(\textbf{let}\ d\ \textbf{in}\ T\big) :_G \tau \mid \Delta \cup \Sigma}\ \textsc{Let}$$

$$\frac{\Gamma \vdash F :_G (n :_t \sigma) \to \rho \mid \Delta \qquad \Gamma \vdash X :_{G \wedge s} \sigma \mid \Sigma}{\Gamma \vdash F\ \widehat{s}\ X :_G \rho[n \mapsto X] \mid \Delta \cup \Sigma \cup t \leftrightarrow s}\ \textsc{App} \qquad \frac{\Gamma \vdash T :_G \tau \mid \Delta \qquad \Gamma \vdash \tau \approx \sigma \mid \Sigma}{\Gamma \vdash T :_G \sigma \mid \Delta \cup \Sigma}\ \textsc{Conv}$$

Fig. 14. Erasure inference rules for $\mathsf{TT}^{\mathsf{evar}}_{\star}$ terms

$$\frac{}{\Gamma \vdash \textsc{Bnds}(\diamond) \mid \emptyset}\ \textsc{BndsBase} \qquad \frac{\Gamma \vdash \textsc{Bnd}(d_1) \mid \Delta \qquad \Gamma, d_1 \vdash \textsc{Bnds}(d_2, \ldots, d_n) \mid \Sigma}{\Gamma \vdash \textsc{Bnds}(d_1, \ldots, d_n) \mid \Delta \cup \Sigma}\ \textsc{BndsStep}$$

$$\frac{b \in \{\textbf{variable}, \textbf{constructor}\} \qquad \Gamma \vdash \tau :_{\{\mathsf{E}\}} \mathsf{Type} \mid \Delta}{\Gamma \vdash \textsc{Bnd}(n :_r \tau = b) \mid \Delta}\ \textsc{BndAbstr}$$

$$\frac{T \text{ is a term} \qquad \Gamma \vdash \tau :_{\{\mathsf{E}\}} \mathsf{Type} \mid \Delta \qquad \Gamma, n :_r \tau \vdash T :_{\{r\}} \tau \mid \Sigma}{\Gamma \vdash \textsc{Bnd}(n :_r \tau = T) \mid \Delta \cup \Sigma}\ \textsc{BndTerm}$$

$$\frac{\Gamma \vdash \tau :_{\{\mathsf{E}\}} \mathsf{Type} \mid \Delta \qquad \forall i.\ \big(\Gamma, n :_r \tau \vdash \textsc{Clause}_r(C_i) \mid \Sigma_i\big)}{\Gamma \vdash \textsc{Bnd}\big(n :_r \tau = \overline{C}\big) \mid \Delta \cup \bigcup_i \Sigma_i}\ \textsc{BndClauses}$$

$$\frac{\begin{array}{c}\Gamma \vdash \textsc{Bnds}(\Pi) \mid \Delta \qquad \Gamma; \Pi \vdash \textsc{PatWF}_f(\lfloor f \rfloor\ \overline{\widehat{s} P}) \qquad \mathrm{FPV}_\Pi\big(\lfloor f \rfloor\ \overline{\widehat{s} P}\big) = \mathrm{BV}(\Pi) \\ \lfloor f \rfloor\ \overline{\widehat{s} P} \text{ is linear} \qquad \lfloor f \rfloor\ \overline{\widehat{s} P} \text{ is forced-pattern-consistent} \\ \Gamma; \Pi \vdash \lfloor f \rfloor\ \overline{\widehat{s} P} :^{r}_{\{r\}} \tau \mid \Lambda \qquad \Gamma, \Pi \vdash R :_{\{r\}} \tau \mid \Sigma\end{array}}{\Gamma \vdash \textsc{Clause}_r(\Pi.\ \lfloor f \rfloor\ \overline{\widehat{s} P} = R) \mid \Delta \cup \Lambda \cup \Sigma}\ \textsc{Clause}$$

Fig. 15. Inference rules for $\mathsf{TT}^{\mathsf{evar}}_{\star}$ bindings

annotations. Constraint set Δ contains constraints equating annotations in the corresponding positions in τ and σ.

4.1.3 Rules. Figure 14 shows the constraint-generating typing rules for $\mathsf{TT}^{\mathsf{evar}}_{\star}$ terms, Figure 15 shows the rules for bindings, and Figure 16 shows the rules for patterns. These inference rules mirror the checking rules, extracting the relationships between erasure annotations in them into constraint sets.

While the checking rules used a single annotation as the subscript on colons in name bindings, the inference rules use a set of annotations there.

$$\frac{(n :_s \sigma = \textbf{variable}) \in \Pi}{\Gamma; \Pi \vdash n :^q_G \sigma \mid s \leftrightarrow G}\ \textsc{PatVar} \qquad \frac{(n :_s \sigma = \textbf{constructor}) \in \Gamma}{\Gamma; \Pi \vdash n :^q_G \sigma \mid q \to (G \wedge s)}\ \textsc{PatCtor}$$

$$\frac{(n :_s \sigma = \textbf{constructor}) \in \Gamma}{\Gamma; \Pi \vdash \lceil n \rceil :^q_G \sigma \mid G \to s}\ \textsc{PatForcedCtor} \qquad \frac{(f :_q \sigma) \in \Gamma}{\Gamma; \Pi \vdash \lfloor f \rfloor :^q_q \sigma \mid \emptyset}\ \textsc{PatDefName}$$

$$\frac{\Gamma; \Pi \vdash F :^q_G (n :_s \sigma) \to \rho \mid \Delta \qquad \Gamma; \Pi \vdash X :^q_{G \wedge t} \sigma \mid \Sigma}{\Gamma; \Pi \vdash F\ \widehat{t}\ X :^q_G \rho[n \mapsto |X|] \mid \Delta \cup \Sigma \cup s \leftrightarrow t}\ \textsc{PatApp}$$

$$\frac{\Gamma, \Pi \vdash T :_G \tau \mid \Delta}{\Gamma; \Pi \vdash \lceil T \rceil :^q_G \tau \mid \Delta}\ \textsc{PatForced} \qquad \frac{\Gamma; \Pi \vdash P :^q_G \tau \mid \Delta \qquad \Gamma, \Pi \vdash \tau \approx \sigma \mid \Sigma}{\Gamma; \Pi \vdash P :^q_G \sigma \mid \Delta \cup \Sigma}\ \textsc{PatConv}$$

Fig. 16. Inference rules for $\mathsf{TT}^{\mathsf{evar}}_{\star}$ patterns

4.2 Solving Constraints

The set of constraints can be interpreted in several ways.

With only two erasure levels, we can view any constraint set Δ as a logic program containing only Horn clauses, where E corresponds to False and R corresponds to True. The set of retained bindings and applications is then given by the minimal model of this logic program, which is unique [Gelfond and Lifschitz 1988; Van Emden and Kowalski 1976] and can be computed in linear time [Dowling and Gallier 1984] in the size of Δ.

We can also view the erasure levels more generally as the lattice $\mathsf{E} < \mathsf{R}$, in which case we can express the solution as a minimal fixed point, and compute it iteratively. This is extensible to lattices containing additional erasure levels, such as irrelevance (not in scope of this paper).

4.2.1 Semantics of Erasure Constraints. More formally, let $\Phi : \mathbb{N} \cup \{\mathsf{R}, \mathsf{E}\} \to \{\mathsf{R}, \mathsf{E}\}$ be a valuation of erasure annotations of $\mathsf{TT}^{\mathsf{evar}}_{\star}$ such that $\Phi(\mathsf{R}) = \mathsf{R}$ and $\Phi(\mathsf{E}) = \mathsf{E}$. We further overload this symbol as $\Phi(G) = \bigwedge_{g \in G} \Phi(g)$ for guard sets. We also overload this symbol for any syntactic construct X of $\mathsf{TT}^{\mathsf{evar}}_{\star}$ to mean substitution for evars, which yields a fully erasure-annotated $\mathsf{TT}^{\mathsf{RE}}_{\star}$ construct $\Phi(X)$. Valuations form a lattice; $\Phi \leq \Phi'$ iff $\forall r.\ \Phi(r) \leq \Phi'(r)$. Valuation of evars Φ is a model of constraint set Δ, written $\Phi \models \Delta$, iff for all constraints $(G \to r) \in \Delta$, $\Phi(G) \leqslant \Phi(r)$. The *solution* of constraint set Δ is its minimal model.

If the program is erasure-inconsistent, no solution will exist. For example, if the program generates constraint $\mathsf{R} \to \mathsf{E}$, the definition of $\models$ requires that $\Phi(\mathsf{R}) \leqslant \Phi(\mathsf{E})$, which is a contradiction with the definition of Φ and the fact that $\mathsf{E} < \mathsf{R}$.

4.3 Correctness

The proofs of the following properties are mostly straightforward. They rely only on a few elementary lemmas, and are much easier than Church-Rosser (Conjecture 1), Subject Reduction (Theorem 2), or commutativity of erasure and reduction (Theorem 1).

Theorem 3 (Soundness of erasure inference)**.** *Any valuation of evars Φ computed by the inference rules is accepted by the checking rules.*

This theorem has four mutually inductive components, one for terms, one for patterns, one for bindings, and one for telescopes of bindings (contexts).

$$\frac{\Gamma \vdash T :_G \tau \mid \Delta \qquad \Phi \models \Delta}{\Phi(\Gamma) \vdash \Phi(T) :_{\Phi(G)} \Phi(\tau)} \qquad \frac{\Gamma \vdash P :^{q}_{G} \tau \mid \Delta \qquad \Phi \models \Delta}{\Phi(\Gamma) \vdash \Phi(P) :^{\Phi(q)}_{\Phi(G)} \Phi(\tau)}$$

$$\frac{\Gamma \vdash \mathit{BND}(d) \mid \Delta \qquad \Phi \models \Delta}{\Phi(\Gamma) \vdash \mathit{BND}\big(\Phi(d)\big)} \qquad \frac{\Gamma \vdash \mathit{BNDS}(\Pi) \mid \Delta \qquad \Phi \models \Delta}{\Phi(\Gamma) \vdash \mathit{BNDS}\big(\Phi(\Pi)\big)}$$

Proof sketch 3. *By (mutual) induction on the typing derivations, using lemmas about basic properties of the system.*

Theorem 4 (Completeness of erasure inference)**.** *Let P be a $TT^{evar}_{\star}$ program. If $\Phi(P)$, which is a $TT^{RE}_{\star}$ program, satisfies the checking rules, then P satisfies the inference rules, and Φ is a model of the corresponding set of constraints (not necessarily optimal).*

Like Theorem 3, also this theorem consists of four mutually recursive components.

$$\frac{\Phi(\Gamma) \vdash \Phi(T) :_{\Phi(G)} \Phi(\tau)}{\exists \Delta. \quad \Gamma \vdash T :_G \tau \mid \Delta \qquad \Phi \models \Delta} \qquad \frac{\Phi(\Gamma) \vdash \Phi(P) :^{\Phi(q)}_{\Phi(G)} \Phi(\tau)}{\exists \Delta. \quad \Gamma \vdash P :^{q}_{G} \tau \mid \Delta \qquad \Phi \models \Delta}$$

$$\frac{\Phi(\Gamma) \vdash \mathit{BND}\big(\Phi(d)\big)}{\exists \Delta. \quad \Gamma \vdash \mathit{BND}(d) \mid \Delta \qquad \Phi \models \Delta} \qquad \frac{\Phi(\Gamma) \vdash \mathit{BNDS}\big(\Phi(\Pi)\big)}{\exists \Delta. \quad \Gamma \vdash \mathit{BNDS}(\Pi) \mid \Delta \qquad \Phi \models \Delta}$$

Proof sketch 4. *By (mutual) induction on the typing derivations, using lemmas about basic properties of the system.*

Corollary 4 (Optimality of erasure inference)**.** *Since erasure inference is complete and the inferred solution is defined as the minimal model, it is optimal among all valuations of the given $TT^{evar}_{\star}$ program that would pass the checking rules.*

4.4 Complexity

If n is the size of the input program and t is the size of its typing derivation, erasure inference is computable within $O(n^2 \log n + t \log n)$ time and $O(n^2)$ space.

- The number of inference rules invoked during erasure inference is t, which includes $O(n)$ invocations of the syntax-driven typing rules, plus a potentially large number of invocations of the conversion/reduction rules.
  We will consider the syntax-driven and conversion/reduction invocations separately.
- Each syntax-driven inference rule performs a *small* number of erasure-inference-related operations from the following palette. A *small* number is at most linear in the number of sub-invocations of inference rules, and thus can be amortised as a constant per invocation.

$$\textbf{let } \mathsf{U} : \mathsf{Type} = \textbf{constructor in}$$
$$\textbf{let } \mathsf{C} :_{r_\mathsf{C}} \mathsf{U} = \textbf{constructor in}$$
$$\textbf{let } \mathsf{F} : \mathsf{U} \to \mathsf{U} \to \mathsf{U} = \textbf{constructor in}$$
$$\underbrace{\mathsf{F}_{\widehat{s_1}}\ \mathsf{C}_{\widehat{r_1}}\ (\mathsf{F}_{\widehat{s_2}}\ \mathsf{C}_{\widehat{r_2}}\ (\mathsf{F}_{\widehat{s_3}}\ \mathsf{C}_{\widehat{r_3}}(\ldots)))}_{n\times}$$

(a) $\mathrm{TT}^{\mathrm{evar}}_{\star}$ program, some evars omitted

$$s_1 \to r_\mathsf{C}$$
$$r_1 \wedge s_2 \to r_\mathsf{C}$$
$$r_1 \wedge r_2 \wedge s_3 \to r_\mathsf{C}$$
$$r_1 \wedge r_2 \wedge r_3 \wedge s_4 \to r_\mathsf{C}$$
$$r_1 \wedge r_2 \wedge r_3 \wedge r_4 \wedge s_5 \to r_\mathsf{C}$$
$$\ldots$$

(b) Generated constraints

Fig. 17. A quadratic case

Operation	Time	Allocated space	Note
$G \to s$	$O(1)$	$O(n)$	$O(1)$ space with sharing
$s \to G$	$O(n)$	$O(n)$	
$r \leftrightarrow s$	$O(1)$	$O(1)$	
$G \wedge s$	$O(\log n)$	$O(\log n)$	assuming tree-based finite sets
$G \leftrightarrow s$	$O(n)$	$O(n)$	
$\Delta \cup \{G \to r\}$	$O(n \log n)$	$O(\log n)$	assuming tree-based finite sets

The table does not contain $\Delta \cup \Sigma$. Instead, we can insert constraints one by one into a big set in a manner similar to the Writer monad. Assuming a tree set, each insertion fits within $O(\log n)$ comparisons of $O(n)$-sized constraints along the insertion path in the tree.

- Each of these operations takes at most $O(n \log n)$ time and allocates at most $O(n)$ new space, which amounts to $O(n^2 \log n)$ time and $O(n^2)$ space for the syntax-driven part of erasure inference.
- The number of non-syntax-driven rules invoked in conversion checking and reduction is not bounded by $O(n)$. However, these rules are already invoked by ordinary type checking and perform only limited erasure-specific work.
  Specifically, erasure inference incurs additional work in rule CRefl, when checking syntactic equality of two terms. Besides the usual equality check required for type-correctness, erasure generates constant-sized constraints, equating pairs of evars in the corresponding places of the two terms.
  There are at most $O(n)$ distinct evars, and thus at most $O(n^2)$ distinct constraints that express equality between pairs of evars. The total size of the generated constraint set, including the constraints produced by the syntax-driven rules, therefore does not exceed $O(n^2)$.
  Since the size of the typing derivation t includes this syntactic equality check, and each rule for constrained equality generates at most two constraints, $\{p \to q, q \to p\}$, the syntactic equality check performs at most $O(t)$ insertions of constant-sized constraints into an $O(n^2)$-sized set of constraints. This can be done in time $O(t \log n)$.
- Worst-case-linear algorithms [Dowling and Gallier 1984] will solve a quadratically sized set of constraints within quadratic time bounds.
- Given the above, we need $O(n^2 \log n)$ time and $O(n^2)$ space for the syntax-driven part of erasure inference, $O(t \log n)$ time and $O(n^2)$ space for the non-syntax-driven part of erasure inference, and $O(n^2)$ time for solving the constraints, which amounts to $O(n^2 \log n + t \log n)$ time and $O(n^2)$ space in total.

Figure 17a shows a program that generates a quadratically sized set of constraints (Figure 17b). It might be possible to use sharing to reduce the size of a concrete representation of this set to linear.

4.5 Error Reporting

Erasure inference can produce error messages for inconsistent annotations by attaching a *reason* to every constraint. Constraints then have the form $G \rightarrow_\rho r$, where ρ is an annotation describing the location in the code and the reason that gave rise to the constraint.

The solver can then keep track of all constraints involved in setting an annotation to R. If any such annotation is manually marked E, the compiler can print a detailed error message explaining why this is an inconsistent annotation.

5 RESULTS

We demonstrate the effectiveness of our erasure method by showing the output of erasure on an example, and by showing a few benchmark results.

We have implemented this erasure method in a small prototype compiler[1], which is itself written in Haskell, accepts programs written in a custom syntax, and generates Chicken Scheme [CHICKEN project 2020]. The generated Scheme code is then compiled to native executables.

This prototype compiler is separate from Idris, although Idris 1 contains an implementation of an untyped, flow-based predecessor of its erasure method.

5.1 Output of Erasure

Let us revisit the binary number data family from the introduction and look at how a small program would be compiled using $\mathsf{TT}_\star$.

Figure 18a shows an elaborated program that defines function add1 and applies it to the binary representation of number 1. This formulation also includes erasure annotations as inferred from a completely unannotated program by erasure inference. The annotations are shown only on the binders, we omit them on application for brevity. This is the fully elaborated form of the program, expressed in the $\mathsf{TT}^{\mathsf{RE}}_\star$ flavour of $\mathsf{TT}_\star$, including all types and erasure annotations, and no unknowns.

By running erasure on this program, which strips all types and E-annotated portions of the program, we obtain the erased program (Figure 18b), expressed in $\mathsf{TT}^{\square}_\star$. This program contains no overhead introduced by dependent typing anymore, and looks like an ordinary functional program.

5.2 Benchmarks

In Figure 19, we present the results of a few benchmarks we have performed with our implementation of erasure.

We ran the benchmarks on a test suite of example programs. The programs range from trivial, such as a single const function, to complex, such as an implementation of a scope-indexed lambda calculus, or a dependent-view-based implementation of a palindrome decider. All example programs are entirely erasure-unannotated, with the exception of places like postulates of FFI printing functions, or equivalent. Here is a brief description of some of them.

foreign is a test for FFI and decidable equality of lists
rev implements the reverse view for lists
rle implements run-length compression and decompression
bin implements an adder of binary numbers
tt implements an evaluator for a scope-indexed lambda calculus
palindrome implements a palindrome decider, using the reverse view as one of its components

Figure 19a illustrates the numbers of annotations in some of the example programs, and how many were inferred as erased or retained. Programs bin and rle have very different sizes, yet

[1] https://github.com/ziman/ttstar

$$
\begin{array}{l}
\textbf{let}\ \mathbb{N} :_{\mathsf{E}} \mathsf{Type} = \textbf{constructor in} \\
\textbf{let}\ \mathsf{Z} :_{\mathsf{E}} \mathbb{N} = \textbf{constructor in} \\
\textbf{let}\ \mathsf{S} :_{\mathsf{E}} (n :_{\mathsf{E}} \mathbb{N}) \to \mathbb{N} = \textbf{constructor in} \\
\textbf{let}\ \mathsf{double} :_{\mathsf{E}} (m :_{\mathsf{E}} \mathbb{N}) \to (n :_{\mathsf{E}} \mathbb{N}) \to \mathbb{N} \\
\quad = \left\{ \begin{array}{llcl} & \lfloor \mathsf{double} \rfloor & \mathsf{Z} & = \mathsf{Z} \\ (n :_{\mathsf{E}} \mathbb{N}). & \lfloor \mathsf{double} \rfloor & (\mathsf{S}\ n) & = \mathsf{S}\ (\mathsf{S}\ (\mathsf{double}\ n)) \end{array} \right\} \textbf{in}
\end{array}
$$

$$
\begin{array}{l}
\textbf{let}\ \mathsf{Bin} :_{\mathsf{E}} (n :_{\mathsf{E}} \mathbb{N}) \to \mathsf{Type} = \textbf{constructor in} \\
\textbf{let}\ \mathsf{N} :_{\mathsf{R}} \mathsf{Bin}\ \mathsf{Z} = \textbf{constructor in} \\
\textbf{let}\ \mathsf{I} :_{\mathsf{R}} (n :_{\mathsf{E}} \mathbb{N}) \to \mathsf{Bin}\ n \to \mathsf{Bin}\ (\mathsf{S}\ (\mathsf{double}\ n)) = \textbf{constructor in} \\
\textbf{let}\ \mathsf{O} :_{\mathsf{R}} (n :_{\mathsf{E}} \mathbb{N}) \to \mathsf{Bin}\ n \to \mathsf{Bin}\ (\mathsf{double}\ n) = \textbf{constructor in}
\end{array}
$$

$$
\begin{array}{l}
\textbf{let}\ \mathsf{add1} : (n :_{\mathsf{E}} \mathbb{N}) \to (x :_{\mathsf{R}} \mathsf{Bin}\ n) \to \mathsf{Bin}\ (\mathsf{S}\ n) \\
\quad = \left\{ \begin{array}{llcccl} & \lfloor \mathsf{add1} \rfloor & [\mathsf{Z}] & \mathsf{N} & = & \mathsf{I}\ \mathsf{Z}\ \mathsf{N} \\ (n :_{\mathsf{E}} \mathbb{N})\ (b :_{\mathsf{R}} \mathsf{Bin}\ n). & \lfloor \mathsf{add1} \rfloor & [\mathsf{double}\ n] & (\mathsf{O}\ n\ b) & = & \mathsf{I}\ n\ b \\ (n :_{\mathsf{E}} \mathbb{N})\ (b :_{\mathsf{R}} \mathsf{Bin}\ n). & \lfloor \mathsf{add1} \rfloor & [\mathsf{S}\ (\mathsf{double}\ n)] & (\mathsf{I}\ n\ b) & = & \mathsf{O}\ (\mathsf{S}\ n)\ (\mathsf{add1}\ n\ b) \end{array} \right\} \textbf{in} \\
\\
\mathsf{add1}\ (\mathsf{S}\ \mathsf{Z})\ (\mathsf{I}\ \mathsf{Z}\ \mathsf{N})
\end{array}
$$

(a) At the $\mathsf{TT}^{\mathsf{RE}}_{\star}$ stage: elaborated, annotated, unerased. Erasure annotations in applications have been omitted for space reasons

$$
\begin{array}{l}
\textbf{let}\ \mathsf{N} = \textbf{constructor in} \\
\textbf{let}\ \mathsf{I} = \textbf{constructor in} \\
\textbf{let}\ \mathsf{O} = \textbf{constructor in} \\
\\
\textbf{let}\ \mathsf{add1} \\
\quad = \left\{ \begin{array}{llcl} \lfloor \mathsf{add1} \rfloor & \mathsf{N} & = & \mathsf{I}\ \mathsf{N} \\ \lfloor \mathsf{add1} \rfloor & (\mathsf{O}\ b) & = & \mathsf{I}\ b \\ \lfloor \mathsf{add1} \rfloor & (\mathsf{I}\ b) & = & \mathsf{O}\ (\mathsf{add1}\ b) \end{array} \right\} \textbf{in} \\
\\
\mathsf{add1}\ (\mathsf{I}\ \mathsf{N})
\end{array}
$$

(b) At the $\mathsf{TT}^{\square}_{\star}$ stage: erased

Fig. 18. A $\mathsf{TT}_{\star}$ program that computes the successor of 1.

they erase to programs of about the same size. There is much more dependent typing involved in program bin.

Figure 19b shows the total runtime of the compiler, including the (external) Chicken Scheme compilation stage. Erasure decreases the amount of code to generate and compile at later stages, and it saves much more time than it takes.

How much time does it take? Figure 19c shows the time taken by erasure inference, depending on the number of erasure annotations. Each data point is an example program from our test suite. We measure the erasure time as the difference between a normal run of the compiler, compared to a run which skips inference and sets all evars to R. The measured time includes all steps such as parsing, erasure inference, and erasure itself, up to and including Scheme code generation. The

times do not include the Scheme compiler stage. It's hard to reasonably infer asymptotic behaviour from this data set but it suggests that our implementation can process about 10^5 erasure annotations per second in our example programs.

We have made no particular effort to make our implementation run fast. Yet in our implementation, when compiling our example programs, erasure inference saves several orders of magnitude more time than it takes.

Figure 19d compares the run time of unerased and erased (compiled) program palin. This program reads n characters as a list, creates a double-ended view of the list, and uses this view to decide whether the list is a palindrome. The plots show the run time on the vertical axis, and the input size n on the horizontal axis. The horizontal axes use different scales in the two plots. We experimentally determined that the unerased version seems to run in $O(n^3)$ time[2], while erasure recovers the $O(n)$ run time[3], as expected.

6 DISCUSSION

6.1 Remarks

6.1.1 Forced Patterns and Erasure Annotations. We claim that the presented erasure method works without any programmer-provided erasure annotations. However, one might argue that forced patterns and forced constructors are a form of erasure annotations.

This is however not true. Erasability is a *consequence* of operational behaviour of a program and the choice of forced patterns is a choice of operational behaviour, not a choice of erasability – we can always have constructor fields that are forced in a pattern match but not erasable, and vice versa. However, if we care about erasability, we need to care about operational behaviour and thus about forced patterns.

This does not mean that programmers have to start annotating forced patterns explicitly if they want good erasure – elaboration in Idris usually does a good job in choosing the forced patterns for the programmer. However, the choice must be made – and it has always been made, tacitly (Idris) or explicitly (Agda) – and since the choice matters, we should give programmers means to influence it.

6.1.2 Modularity of the Analysis. The erasure inference procedure, as given in this paper, is whole-program. Real-world implementations will need to modularise it. Our experience suggests that a useful approach might be requiring programmers to erasure-annotate data constructors explicitly, and solving erasure constraints for everything else automatically per-module.

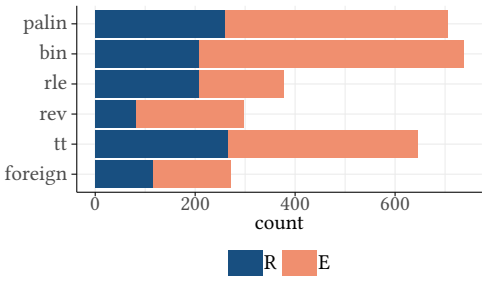
6.1.3 Matching on Erased Data. The rules of $\mathsf{TT}_\star$ allow matching on erased data under certain circumstances. This has already been shown in Section 2.3, where we project variable k out of an erased argument.

Another very common example of this is matching on equality proofs. Consider the function subst shown in Figure 20. In the (only) clause of subst, Refl is a forced constructor because it is the only possible choice, consistent according to Section 3.5.3, Definition 1. We furthermore choose to force its argument x by the implicit argument x of function subst.

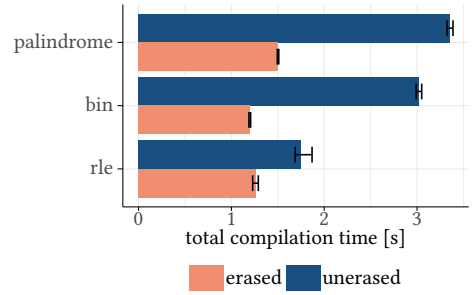
Since the tag and the only field of Refl is erased, erasure inference will then mark the equality proof (and most other arguments of subst) as erased, and subst becomes an identity function after erasure.

[2]We cannot explain why the experimentally determined exponent is approximately 2.97, while the program should be quadratic in theory.

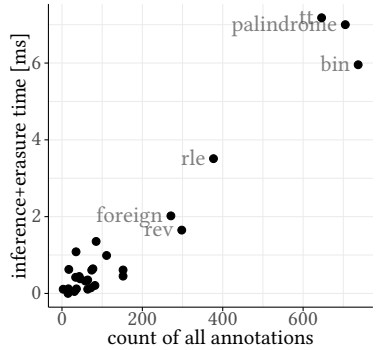
[3]The peaks in the plot seem to be exponentially spaced and we expect that they are caused by Chicken RTS effects, such as more frequent garbage collection when approaching a heap resizing threshold.



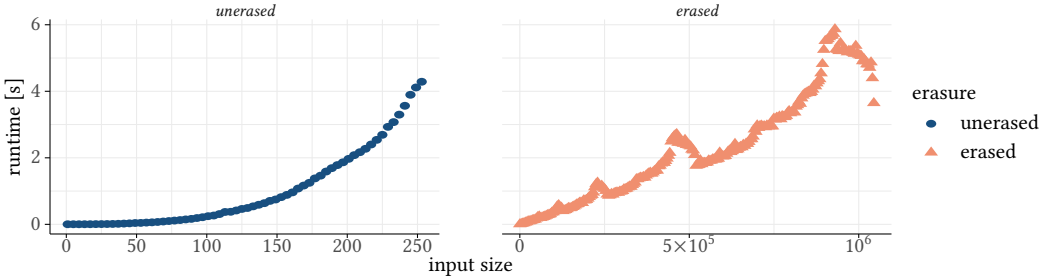
(a) Inferred annotations



(b) Total compilation time including erasure
Error bars show minimum and maximum around the mean.



(c) Erasure time vs. number of annotations



(d) Runtime of a typical program (palindrome decider), from $O(n^3)$ to $O(n)$.

Fig. 19. Erasure in numbers and plots
Most plots show means over multiple runs.
Error bars that have been omitted would be uselessly narrow.

The example in Section 2.3 further illustrates that the notion of erasure is much more subtle than just disallowing matching on erased data, and it may take into account also contextual clues and inductive invariants.

```
data Id : {a : Type} → (x : a) → (y : a) → Type where
  Refl : {a : Type} → (x : a) → Id {a} x x

subst : {a : Type} → {P : a → Type} → {x : a} → {y : a}
  → Id {a} x y → (px : P x) → P y
subst {a} {P} {x} {y} (⌈Refl⌉ [x]) px = px
```

Fig. 20. Pattern-matching on erased arguments

6.1.4 Identity Optimisation. Thanks to erasure, we can automatically spot identity functions post-erasure. Our implementation has a separate pass that removes applications of identity functions, such as subst above, from the erased code to make it even cleaner.

This brings further asymptotic improvements since it can also recognise various weakening functions such as finWeaken : Fin n → Fin (S n), which also become identity functions after erasure, thus eliminating $O(n)$ work in each such application.

We do not discuss further details of this optimisation in this paper; we just note that this optimisation is an indirect benefit of erasure.

6.1.5 Strong Normalisation. It is known that unrestricted erasure does not preserve strong normalisation [Letouzey 2003, 2004, 2008; Mishra-Linger 2008]. Mishra-Linger shows a brief example [2008, p. 133], where a function takes a void argument, which ends up being erased. This exposes ill-typed redexes beneath the erased lambda even without providing the (impossible) argument.

Letouzey observes that these "problematic" invocations are rather rare [Letouzey 2008, Sec. 4] and the extraction process of Coq is special-cased to leave problematic terms guarded by at least one lambda – by erasing *all* erasable arguments of a function, and by inserting dummy lambdas around problematic partial applications [Letouzey 2003, Sec. 2.1].

TT$_\star$ does not address this problem explicitly and does not discuss totality checking, either; its implementations may prefer to.

6.1.6 FFI and Monadic I/O. TT$_\star$ works well with FFI and monadic I/O. For FFI, the programmer must erasure-annotate foreign postulates explicitly, and everything else is inferred. Monadic I/O needs exactly one well-placed R-annotation in the standard library to prevent the compiler from erasing the RealWorld token. The token is indeed unused but we want to retain it to obtain proper delaying and sequencing of side effects.

6.2 Related Work

The literature approaches the question of erasure in various ways, and these approaches could be classified by various criteria, such as whether irrelevance is identified with erasure or whether erasure is a property of variable bindings (extrinsic erasure) or mediated by the type of variables (intrinsic erasure); whether erasure is identified with irrelevance, etc.

6.2.1 Mishra-Linger, IPTS, EPTS. TT$_\star$ uses an approach similar to Mishra-Linger's dissertation [2008], in that erasure is extrinsic: it's a property of the context in which a term appears, not of its type. Mishra-Linger also does not conflate irrelevance and erasability, but discusses the possibility of doing so. TT$_\star$ adds mainly the support for type families and full dependent pattern matching.

6.2.2 Plenty O' Nuttin'. Like EPTS [Mishra-Linger 2008] and TT$_\star$, McBride's calculus [2016] introduces quantities on binders. Unlike TT$_\star$, the calculus allows an extra "linear" quantity in addition to "erased" and "unerased", thus unifying erasure and linear types in a single dependently typed

framework. It is presented bidirectionally and does not annotate applications with quantities. The calculus uses eliminators instead of pattern matching, in line with McBride's previous work [Goguen et al. 2006]. There is no inference of quantities.

6.2.3 QTT. Atkey [2018] reformulates and corrects McBride's calculus [2016], developing its semantics. QTT does not feature the rule Weak introduced in McBride's calculus. Atkey does not give pattern matching or inference of quantities, either.

6.2.4 Idris. $\mathsf{TT}_\star$ is based on TT, the core calculus of Idris [Brady 2013], although it departs from Idris in how data is represented. Idris originally had a way to erase some data via forcing, detagging, and collapsibility [Brady et al. 2004]. This method cannot erase indices like those occurring in the type family of binary numbers in this paper, as discussed in Section 2.4.

Later development added untyped, flow-based erasure inference, which can erase the examples shown in this paper – and was a significant improvement on the status quo in 2014 – but does not work well in higher-order and very dependent scenarios. Consider the following (contrived but short) program.

$$
\begin{array}{l}
\mathsf{RetTy} : (\mathit{erased} : \mathsf{Bool}) \to \mathsf{Type} \\
\mathsf{RetTy}\quad \mathsf{True}\quad =\quad (x :_\mathsf{E} \mathsf{Int}) \to \mathsf{Int} \\
\mathsf{RetTy}\quad \mathsf{False}\quad =\quad (x :_\mathsf{R} \mathsf{Int}) \to \mathsf{Int}
\end{array}
\qquad
\begin{array}{l}
\mathsf{f} : (\mathit{erased} : \mathsf{Bool}) \to \mathsf{RetTy}\ \mathit{erased} \\
\mathsf{f}\quad \mathsf{True}\quad x\quad =\quad 0 \\
\mathsf{f}\quad \mathsf{False}\quad x\quad =\quad x
\end{array}
$$

There is little hope to express the above erasure pattern without typed erasure: the second argument of f can be erased if its first argument is True. $\mathsf{TT}_\star$ does not have this problem and it can spot the erasure opportunity even if the explicit annotations in RetTy are removed.

Idris 2 [Brady 2020] has a new core calculus based on QTT [Atkey 2018], and thus features typed erasure like $\mathsf{TT}_\star$, although without erasure inference.

6.2.5 Agda. Agda [Norell 2007] has multiple ways to achieve erasure. The forcing optimisation [Brady 2005], can erase some interesting classes of data (Section 2.4). Irrelevance [Agda authors 2020a] is sufficient for erasure of proofs, especially proof members of records, but does not work for erasure of indices, as discussed in Section 2.2.

Shape Irrelevance. To alleviate some issues with full-on irrelevance in a language with typed equality and η-expansion, Agda features *shape irrelevance* [Abel et al. 2017]. The programmer can mark some arguments of type constructors as shape-irrelevant. Such a type constructor can then receive irrelevant arguments in these positions. In practice, this results in sizes ignored in the equality of (applications of) data constructors, because they are irrelevant there, but not ignored in the equality of (applications of) type constructors, because they are only shape-irrelevant there. In the literature, shape-irrelevance works only for sizes in the context of Agda's sized types, and cannot be consistently generalised for all types [Abel et al. 2017], but could probably cover the most useful use cases. In the implementation, Agda experimentally allows shape-irrelevant indices for non-Size types under certain circumstances, too.

Run-Time Irrelevance. A relatively recent addition to Agda is *run-time irrelevance* [Agda authors 2020b], which seems to be equivalent to erasure as discussed in this paper.

6.2.6 Zombie. Zombie [Sjöberg 2015] also uses irrelevance to express erasure. Irrelevance is extrinsic. With untyped equality, this does not seem to cause serious issues, except for the mild loss of expressivity caused by the identification of the two notions. Zombie can successfully erase the examples in this paper. Zombie has parameterised inductive types and non-dependent pattern matching. Indexed type families are encoded indirectly using Henry Ford indices [McBride 2000],

and dependencies from pattern matching are reified as explicit equalities in the scope. $\mathsf{TT}_\star$ supports dependent pattern matching directly and has erasure inference.

6.2.7 Cedille. Cedille [Stump 2018] has the implicit product type $\forall x : T.\ T'$, which is constructed by the implicit lambda $\Lambda x : T.\ T'$, similar to ICC [Miquel 2001]. Cedille also supports erased let expressions. Erased values are irrelevant in Cedille, and equality is untyped. Inductive families and pattern matching are expressed in terms of induction principles in the core language. There is no inference of erasure.

6.2.8 Dafny and Boogie 2. Dafny [Leino 2010] is an imperative language with built-in specification constructs, and semantics specified by translation to Boogie 2 [Leino and Rümmer 2010]. Dafny features *ghost variables*, where "as far as the verifier is concerned, there is no difference between ghost variables and physical variables" [Leino 2010]. Flow of information from ghost to non-ghost variables is syntactically forbidden and thus the compiler may erase ghost variables from the program before generating code.

6.2.9 Why3. Why3 [Filliâtre and Paskevich 2013] is a platform for program verification. Its programming and specification language, WhyML, features ghost values, expressions, and constructor fields, which are erased during code extraction.

6.2.10 Dependent Haskell. Dependent Haskell [Eisenberg 2016; Gundry 2013; Weirich et al. 2017] lets the programmer choose from a palette of 12 quantifiers, each offering a different combination of dependence, erasability[4], visibility and matchability [Eisenberg 2016, Sec. 4.2.5, Fig. 4.1]. In the absence of explicit choice, the default is the usual non-dependent, unerased, visible binder ($\to$). $\mathsf{TT}_\star$ models only the erasure aspect, and has inference for it.

6.2.11 Coq. Coq [The Coq development team 2004] achieves erasure via a separate universe of types, called Prop. As the name suggests, the universe is intended for propositions, proofs of which can be erased. Values of non-propositional types, such as the indices of the binary type family shown in this paper, cannot be erased this way, as discussed in Section 2.1.

There have been attempts to bring erasability of non-proofs to Coq without duplicating all non-Prop types in Prop [Leivent 2014a; Letouzey and Spitters 2005] via an erased monad. However, the erased monad in Coq has shortcomings.

Size Issues. Dockins pointed out [2014] that the axiom $(\mathsf{Erase}\ x = \mathsf{Erase}\ y) \to (x = y)$ is inconsistent if $\mathsf{Erased} : \mathsf{Type} \to \mathsf{Prop}$, by reduction to Hurkens's paradox [Hurkens 1995]. Restricting Erased to $\mathsf{Set} \to \mathsf{Prop}$ may be a way to resolve this issue [Herbelin 2014], but then erasure cannot talk about type values.

Indirectness / Syntactic Noise. Manipulation of values in the Erased monad requires pushing operations and values into the monad. In contrast, one can simply apply any pure function to an extrinsically erased value without any ceremony, as long as the whole application appears in an erased position.

Contextuality and Interactive Development. Consider the following incomplete definition of a function that should take an erased natural number and return an erased pair of natural numbers, expressed in an Idris-style syntax.

$$f : \mathsf{Erased}\ \mathbb{N} \to \mathsf{Erased}\ (\mathbb{N}, \mathbb{N})$$
$$f\ en = \mathsf{Erase}\ (\__0, \__1)$$

[4]Eisenberg calls this property "irrelevance".

Above, $\__0$ and $\__1$ are *holes*, which are hypothetical values that would complete the program. Type-based erasure will disallow filling the holes with the erased natural number *en* because the types of the holes are non-erased, even though the whole pair is erased.

It may therefore happen during interactive proving that the programmer faces an unsolvable proof obligation, where they need to supply an unerased value while having only an erased value in the context, even though the overall goal is erased. In such cases, they need to backtrack and redo the proof while taking care to manually preserve the erasedness of the goal along the way.

The same restriction is present in type signatures, which are themselves always entirely erased.

This shows that type-based erasure is inconvenient in these situations: it does not make use of the contextual information.

Erasure Inference. Upon discovery of erasability of a bound variable, erasure inference would have to change its type from τ to Erased τ. Furthermore, due to the indirectness mentioned above, it would also have to rewrite all terms that depend on this variable, including at the type level, because of the lack of contextuality mentioned above. This changes the type signature of the function and further rewriting may be necessary in its transitive dependencies and transitive reverse dependencies. This is likely not practical.

6.2.12 F.* F* has a special effect monad called GHOST, which is an abstract alias of PURE, with automatic lifting of PURE computations to GHOST at the type level (but never in the other direction). [Swamy et al. 2016]

F* has another abstract alias, erased, defined as the identity type mapping **private type** erased $(a$: Type$) = a$. Since this definition is hidden, user code sees erased a as an opaque type distinct from a, and cannot interchange the two.

Type erased a is opaque but its values can be "unwrapped" in the GHOST monad using reveal : erased $a \to$ G a, (G is the effect connected with the GHOST monad), and thus available in specifications. Hence the name *translucent abstraction*: not entirely opaque, not entirely transparent; one can get only a ghostly glimpse of what the erased value once used to be.

There is work on inference of effects [Ahman et al. 2016] but it omits the GHOST monad.

6.3 Future Work

Church-Rosser. Confluence of reduction in $\mathsf{TT}_\star$ is currently a conjecture. The authors have a partial proof, which needs generalisation due to the power of unrestricted **let**. Future work might need to approach **let** a bit less ambitiously.

Erasure Polymorphism of Functions. Consider the program in Figure 21. In the expression (id 3 +

$$
\begin{array}{ll}
\mathsf{expensive} : \mathbb{N} & \mathsf{apply} : \big((x : \mathbb{N}) \to \mathbb{N}\big) \to (y : \mathbb{N}) \to \mathbb{N} \\
\mathsf{expensive} = \ldots & \mathsf{apply}\ f\ y = f\ y \\
 & \\
\mathsf{const42} : (x : \mathbb{N}) \to \mathbb{N} & \mathsf{id} : (x : \mathbb{N}) \to \mathbb{N} \\
\mathsf{const42}\ x = 42 & \mathsf{id}\ x = x
\end{array}
$$

Fig. 21. Example of a program where erasure polymorphism would be useful.

const42 expensive), erasure inference will correctly notice that const42 does not use its argument, marking the expensive argument as erased.

However, in (apply id 3 + apply const42 expensive), erasure inference will not recognise expensive as erased, even though the program is "morally" the same as the previous one without apply.

This is not a shortcoming of *inference*. The problem is that there is no $\mathsf{TT}_\star$ type of apply that would lead to erasure of expensive in this program while keeping it consistent, even if we annotate it manually. Besides unnecessary evaluation (depending on the normalisation strategy), non-erasure of expensive might also (transitively) block erasure in other parts of the program.

Erasure Polymorphism of Data. Both Idris and Coq have several flavours of dependent pairs, differing in which component(s) of the pair are erased, and programmers must choose the correct flavour explicitly. It would be useful to extend the erasure framework to deal with this automatically, for all data types, without explicit duplication or explicit usage of the Erased wrapper.

More Quantities. We are working at merging the ideas from $\mathsf{TT}_\star$, quantity-based linear systems [Atkey 2018; McBride 2016], and irrelevance into a single system with four modalities/quantities (irrelevant, erased, linear/affine, unrestricted), quantity inference, and pattern matching. This seems to work but $\mathsf{TT}_\star$-style unrestricted whole-program inference of linearity seems to be computationally challenging in larger programs since the constraints are no longer monotonic. A more conservative approach may be necessary.

6.4 Conclusion

If we want to propose dependently typed languages as a way to reduce the cost of correctness in programming, we must address the inefficiencies that dependent types could cause. Making programs run *asymptotically* slower just by describing them more precisely is unacceptable.

In this paper, we argued why erasure is necessary (Section 1); shown why some of the existing approaches are insufficient and address related but different problems (Section 2); proposed a core calculus that supports erasure (Section 3) while affording conveniences such as pattern matching; shown that erasure has desirable properties such as preservation of reduction (Section 3.7); proposed an erasure inference algorithm that infers annotations in partly or completely unannotated programs (Section 4); showed that this inference algorithm is sound, complete, and optimal with respect to the typing rules (Section 4.3); and showed using an implementation and its benchmarks that the erasure approach is effective in that it may not only make the compiled programs *asymptotically* faster at run time, and enable other optimisations that further improve the asymptotic complexity of the generated code (Section 6.1.4), but in our implementation, erasure saves several orders of magnitude more compilation time than it takes.

We therefore believe that erasure is a win-win solution to the above unacceptable situation.

ACKNOWLEDGMENTS

The author would like to thank Edwin Brady, Andreas Abel, Jonathan Leivent, and Jan de Muijnck-Hughes for feedback, conversations, suggestions, draft reviews, and other support.

The author is very grateful to the anonymous referees for their reviews, which improved many aspects of the paper.

This work has been supported by the University of St Andrews, School of Computer Science.

REFERENCES

Andreas Abel. 2011. *Irrelevance in Type Theory with a Heterogeneous Equality Judgement.* Springer Berlin Heidelberg, Berlin, Heidelberg, 57–71. https://doi.org/10.1007/978-3-642-19805-2_5

Andreas Abel. 2017. Irrelevance and resurrection in type signatures. Agda Mailing List, 29 July 2017. https://lists.chalmers.se/pipermail/agda/2017/009640.html

Andreas Abel, Andrea Vezzosi, and Theo Winterhalter. 2017. Normalization by evaluation for sized dependent types. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 33. https://doi.org/10.1145/3110277

Agda authors. 2020a. *Agda 2.6.1 documentation: Irrelevance.* https://agda.readthedocs.io/en/v2.6.1/language/irrelevance.html Accessed on 23 May 2020.

Agda authors. 2020b. *Agda 2.6.1 documentation: Run-time Irrelevance.* https://agda.readthedocs.io/en/v2.6.1/language/runtime-irrelevance.html Accessed on 23 May 2020.

Danel Ahman, Catalin Hritcu, Guido Martínez, Gordon D. Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. 2016. Dijkstra Monads for Free. *CoRR* abs/1608.06499 (2016). https://doi.org/10.1145/3093333.3009878 arXiv:1608.06499

Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 56–65. https://doi.org/10.1145/3209108.3209189

Bruno Barras and Bruno Bernardo. 2008. The Implicit Calculus of Constructions As a Programming Language with Dependent Types. In *Proceedings of the Theory and Practice of Software, 11th International Conference on Foundations of Software Science and Computational Structures (FOSSACS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 365–379. https://doi.org/10.1007/978-3-540-78499-9_26

Edwin Brady. 2005. *Practical Implementation of a Dependently Typed Functional Programming Language.* Ph.D. Dissertation. http://eb.host.cs.st-andrews.ac.uk/writings/thesis.pdf

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

Edwin Brady. 2020. Idris 2: Quantitative Type Theory in Action. Submitted. https://www.type-driven.org.uk/edwinb/papers/idris2.pdf

Edwin Brady, Conor McBride, and James McKinna. 2004. Inductive families need not store their indices. In *Types for Proofs and Programs, Torino, 2003, volume 3085 of LNCS*. Springer-Verlag, 115–129. https://doi.org/10.1007/978-3-540-24849-1_8

The CHICKEN project. 2020. CHICKEN Scheme. https://www.call-cc.org/. Accessed: 2020-04-25.

Robert Dockins. 2014. Re: Problem with tactic-generated terms. https://sympa.inria.fr/sympa/arc/coq-club/2014-09/msg00114.html, accessed on 2015-02-28.

William F. Dowling and Jean H. Gallier. 1984. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *The Journal of Logic Programming* 1, 3 (1984), 267 – 284. https://doi.org/10.1016/0743-1066(84)90014-1

Richard A Eisenberg. 2016. *Dependent types in Haskell: Theory and practice.* Ph.D. Dissertation. University of Pennsylvania.

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 – Where Programs Meet Provers. In *ESOP'13 22nd European Symposium on Programming (LNCS)*, Vol. 7792. Springer, Rome, Italy. https://doi.org/10.1007/978-3-642-37036-6_8

Michael Gelfond and Vladimir Lifschitz. 1988. The stable model semantics for logic programming.. In *ICLP/SLP*, Vol. 88. 1070–1080.

Healfdene Goguen, Conor McBride, and James McKinna. 2006. *Eliminating Dependent Pattern Matching.* Springer Berlin Heidelberg, Berlin, Heidelberg, 521–540. https://doi.org/10.1007/11780274_27

Adam Gundry. 2013. *Type Inference, Haskell and Dependent Types.* Ph.D. Dissertation. University of Strathclyde. http://adam.gundry.co.uk/pub/thesis/

Hugo Herbelin. 2014. Re: Problem with tactic-generated terms. https://sympa.inria.fr/sympa/arc/coq-club/2014-09/msg00118.html, accessed on 2015-02-28.

Antonius J. C. Hurkens. 1995. A simplification of Girard's paradox. In *Typed Lambda Calculi and Applications*, Mariangiola Dezani-Ciancaglini and Gordon Plotkin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 266–278. https://doi.org/10.1007/BFb0014058

K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

K Rustan M Leino and Philipp Rümmer. 2010. A polymorphic intermediate verification language: Design and logical encoding. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 312–327. https://doi.org/10.1007/978-3-642-12002-2_26

Jonathan Leivent. 2014a. Mindless Coding. https://github.com/jonleivent/mindless-coding, accessed on 2015-02-28.

Jonathan Leivent. 2014b. Re: Problem with tactic-generated terms. https://sympa.inria.fr/sympa/arc/coq-club/2014-09/msg00110.html, accessed on 2015-02-28.

Pierre Letouzey. 2003. A New Extraction for Coq. In *Types for Proofs and Programs*, Herman Geuvers and Freek Wiedijk (Eds.). Lecture Notes in Computer Science, Vol. 2646. Springer Berlin Heidelberg, 200–219. https://doi.org/10.1007/3-540-39185-1_12

P. Letouzey. 2004. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq.* Ph.D. Dissertation. Université Paris-Sud.

P. Letouzey. 2008. Extraction in Coq, an Overview. In *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008 (Lecture Notes in Computer Science)*, A. Beckmann, C. Dimitracopoulos, and B. Löve (Eds.), Vol. 5028.

Springer-Verlag. https://doi.org/10.1007/978-3-540-69407-6_39

Pierre Letouzey and Bas Spitters. 2005. Implicit and noncomputational arguments using monads.

Conor McBride. 2000. *Dependently typed functional programs and their proofs.* Ph.D. Dissertation. University of Edinburgh. College of Science and Engineering. School of Informatics.

Conor McBride. 2016. I got plenty o'nuttin'. In *A List of Successes That Can Change the World.* Springer, 207–233. https://doi.org/10.1007/978-3-319-30936-1_12

Alexandre Miquel. 2001. The Implicit Calculus of Constructions. In *TLCA*. 344–359. https://doi.org/10.1007/3-540-45413-6_27

Richard Nathan Mishra-Linger. 2008. Irrelevance, Polymorphism, and Erasure in Type Theory. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.154.5619&rep=rep1&type=pdf

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory.* Ph.D. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

Vilhelm Sjöberg. 2015. *A Dependently Typed Language with Nontermination.* Ph.D. Dissertation. University of Pennsylvania.

Aaron Stump. 2018. Syntax and Semantics of Cedille. *CoRR* abs/1806.04709 (2018). arXiv:1806.04709 http://arxiv.org/abs/1806.04709

Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. 2016. Dependent types and multi-monadic effects in F. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 256–270. https://doi.org/10.1145/2837614.2837655

The Coq development team. 2004. *The Coq proof assistant reference manual.* LogiCal Project. http://coq.inria.fr Version 8.0.

Maarten H Van Emden and Robert A Kowalski. 1976. The semantics of predicate logic as a programming language. *Journal of the ACM (JACM)* 23, 4 (1976), 733–742. https://doi.org/10.1145/321978.321991

Stephanie Weirich, Antoine Voizard, Pedro Henrique Azevedo de Amorim, and Richard A. Eisenberg. 2017. A Specification for Dependent Types in Haskell. *Proc. ACM Program. Lang.* 1, ICFP, Article 31 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110275