Practical Erasure in Dependently Typed Languages

Matúš Tejiščák Edwin Brady
University of St Andrews
{mt65,ecb10}@st-andrews.ac.uk

Abstract

Full-spectrum dependently typed languages and tools, such as Idris and Agda, have recently been gaining interest due to the expressive power of their type systems, in particular their ability to describe precise properties of programs which can be verified by type checking.

With full-spectrum dependent types, we can treat types as *first-class* language constructs: types can be parameterised on values, and types can be computed like any other value. However, this power brings new challenges when compiling to executable code. Without special treatment, values which exist only for compile-time checking may leak into compiled code, even in relatively simple cases. Previous attempts to tackle the problem are unsatisfying in that they either fail to erase all irrelevant information, require user annotation or in some other way restrict the expressive power of the language.

In this paper, we present a new erasure mechanism based on whole-program analysis, currently implemented in the Idris programming language. We give some simple examples of dependently typed functional programs with compile-time guarantees of their properties, but for which existing erasure techniques fall short. We then describe our new analysis method and show that with it, erasure can lead to asymptotically faster code thanks to the ability to erase not only proofs but also indices.

1. Introduction

Dependent types give significant expressive power to a programming language. *Full-spectrum* dependent types, as implemented in Idris [3] and Agda [17], treat types as first-class language constructs. This means that types can be built by computation just like any other value, leading to powerful techniques for generic programming. Furthermore, it means that types can be parameterised on values, meaning that strong, explicit, checkable relationships can be stated between values and used to verify properties of programs at compile-time. This expressive power brings new challenges, however, when compiling programs. In principle, extra static information ought to make compiling efficient code *easier*, because the compiler has more information to work with. In practice, however, without special treatment it not only makes programs *slower*, but also have worse *time and space complexity*. While the former is perhaps tolerable in the interests of strong correctness guarantees, the latter is clearly unacceptable!

To illustrate the problem, consider a program in Idris which compresses a list of characters by run-length encoding – that is, we store the list as characters paired with a number of repetitions, such that e.g. "aaaabbbbb" is stored as 4 ‘a‘, 5 ‘b‘. (Note that for brevity, we typeset character lists as string literals.) We can express the relationship between a compressed list, and the original list, by parameterising the compressed form over the original list:

$$\begin{aligned}&\textbf{data}\ \mathsf{RLE} : \mathsf{List\ Char} \to \star\ \textbf{where}\\ &\quad \mathsf{REnd} \ : \ \mathsf{RLE}\ [\,]\\ &\quad \mathsf{RChar} \ : \ (n : \mathbb{N}) \to (c : \mathsf{Char}) \to (rs : \mathsf{RLE}\ xs)\\ &\qquad\qquad \to \mathsf{RLE}\ (\mathsf{replicate}\ n\ c +\!\!+ xs)\end{aligned}$$

The type states that if a compressed list begins with n copies of c, followed by a compressed list rs, then the original list must take the form $\mathsf{replicate}\ n\ c +\!\!+ xs$. As a result, any implementation of an encoder into this form guarantees to produce a *sound* (but not necessarily unique) encoding:

$$\mathsf{rle} : (xs : \mathsf{List\ Char}) \to \mathsf{RLE}\ xs$$

Then, the compressed form of the list "aaaabbbbb" might be represented as:

$$\mathsf{RChar}\ 4\ \text{‘a‘}\ (\mathsf{RChar}\ 5\ \text{‘b‘}\ \mathsf{REnd})$$

If, however, we look closely at the constructors of the RLE family, we notice a potential problem. There is an implicit argument xs in the type of RChar, required for type checking:

$$\begin{aligned}\mathsf{RChar} \ : \ &\{xs : \mathsf{List\ Char}\} \to (n : \mathbb{N}) \to (c : \mathsf{Char})\\ &\to (rs : \mathsf{RLE}\ xs) \to \mathsf{RLE}\ (\mathsf{replicate}\ n\ c +\!\!+ xs)\end{aligned}$$

So, internally, the compressed form of the list "aaaabbbbb" above would really be represented as:

$$\mathsf{RChar}\ \text{“bbbbb”}\ 4\ \text{‘a‘}\ (\mathsf{RChar}\ \text{“”}\ 5\ \text{‘b‘}\ \mathsf{REnd})$$

In other words, all but the prefix of the list needs to be stored in its uncompressed form! And, in fact, the problem gets worse: because the index of RChar is a function, building this implicit value may result in computation at run-time which is ultimately unnecessary.

It gets *even worse* when we look at the full type signature of the decompression function:

$$\mathsf{unrle} : \{xs : \mathsf{List\ Char}\} \to \mathsf{RLE}\ xs \to \mathsf{List\ Char}$$

In order to decompress a RLE-encoded list, in the internal representation, the caller must also provide the decompressed list in the implicit argument!

1.1 Contributions

The challenge, in short, is to identify a *phase distinction* between compile-time and run-time objects. Traditionally, this is simple:

types are compile-time only, *values* are run-time, and erasure consists simply of erasing types. In a dependently typed language, however, erasing types alone is not enough. In this paper, we present an analysis which calculates the phase distinction for dependently typed programs, making the following principal contributions:

- We present a *whole program* analysis which identifies the parts of functions and data structures which are *irrelevant* to the result produced by the main program, and justify its soundness.
- We describe a core language consisting of bindings and case trees on which the analysis can be applied, which is not specific to Idris and can therefore be used to implement erasure analysis for other languages.
- We demonstrate the effectiveness of the erasure analysis with several examples, showing that it not only reduces run-time (and in some cases, time and space complexity) but also that it (perhaps surprisingly) reduces compile times.

Unlike previous approaches to the problem, our analysis can erase not only proofs but also indices. It is entirely automatic; no user annotations are required and no expressive power is lost. In the rest of this paper, we will describe the core calculus on which the analysis is applied (Section 3), the algorithm itself (Section 4), justify its soundness (Section 5) and present benchmarks for both compile time and run time (Section 6).

2. Motivating Examples

In this section, we present two small example programs, typical of the kind of programs we write in Idris but nevertheless demonstrating the difficulties encountered in identifying a phase distinction. Additionally, we briefly discuss alternative approaches to erasure analysis and discuss where they fall short.

2.1 Palindromes

Full dependent types allow us to express facts about data, such as the following predicate, where Palindrome xs expresses that the list xs is palindromic:

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Palindrome} : \mathsf{List}\ a \to \star\ \textbf{where} \\
\quad \begin{array}{rcl}
\mathsf{PNil} & : & \mathsf{Palindrome}\ [\,] \\
\mathsf{POne} & : & (x : a) \to \mathsf{Palindrome}\ [x] \\
\mathsf{PTwo} & : & (x : a) \to (u : \mathsf{Palindrome}\ xs) \\
 & & \quad \to \mathsf{Palindrome}\ (x :: xs +\!\!+ [x])
\end{array}
\end{array}
$$

When we write a function which tests whether a list is palindromic, rather than writing boolean we can return an instance of this predicate, if it holds:

$$
\begin{array}{l}
\mathsf{isPalindrome} : \mathsf{DecEq}\ a \\
\quad \Rightarrow (xs : \mathsf{List}\ a) \to \mathsf{Maybe}\ (\mathsf{Palindrome}\ xs)
\end{array}
$$

Here, DecEq is a type class constraint which expresses that the parameter a has a *decidable equality*. In fact, Palindrome is a special case of a "U-list", which is a *view* [15] of lists giving access to the first and last elements:

$$
\begin{array}{l}
\textbf{data}\ \mathsf{U} : \mathsf{List}\ a \to \star\ \textbf{where} \\
\quad \begin{array}{rcl}
\mathsf{Nil} & : & \mathsf{U}\ [\,] \\
\mathsf{One} & : & (x : a) \to \mathsf{U}\ [x] \\
\mathsf{Two} & : & (x : a) \to (u : \mathsf{U}\ xs) \to (y : a) \\
 & & \quad \to \mathsf{U}\ (x :: xs +\!\!+ [y])
\end{array}
\end{array}
$$

A view gives an alternative way of matching on a structure, using the type to preserve the link with the original structure. It is then easy to write *isPalindrome* as a *total* function by writing a covering function for this view, toU, then checking corresponding first and last elements, combining the following two functions:

$$
\begin{array}{rcl}
\mathsf{toU} & : & (xs : \mathsf{List}\ a) \to \mathsf{U}\ xs \\
\mathsf{isPalinU} & : & \mathsf{DecEq}\ a \Rightarrow \{xs : \mathsf{List}\ a\} \to \mathsf{U}\ xs \\
 & & \quad \to \mathsf{Maybe}\ (\mathsf{Palindrome}\ xs)
\end{array}
$$

Unfortunately, while bringing out this structure explicitly leads to a clean definition, looking at the type of Two shows that there is an overhead:

$$
\begin{array}{rcl}
\mathsf{Two} & : & \{a : \star\} \to \{xs : \mathsf{List}\ a\} \\
 & & \quad \to (x : a) \to (u : \mathsf{U}\ xs) \to (y : a) \\
 & & \quad \to \mathsf{U}\ (x :: xs +\!\!+ [y])
\end{array}
$$

The original lists are present in the structure, as well as the U-list, and without special treatment, the $O(n)$-sized index xs will be recomputed and stored repeatedly in each of the $O(n)$ steps while running isPalindrome. The U-abstraction comes at a significant cost: increasing the time and memory complexity of any algorithm using it.

2.2 Binary Adder

We can define unary natural numbers as a data type in Idris, with an addition function defined by pattern matching and recursion:

$$
\begin{array}{l}
\textbf{data}\ \mathbb{N} = \mathsf{Z} \mid \mathsf{S}\ \mathbb{N} \\
\\
\begin{array}{lrcl}
(+) : \mathbb{N} \to \mathbb{N} \to \mathbb{N} & & & \\
(+) & \mathsf{Z} & y & = & y \\
(+) & (\mathsf{S}\ k) & y & = & \mathsf{S}\ (k + y)
\end{array}
\end{array}
$$

Indeed, these are defined as part of the Idris prelude, imported in every Idris program by default. Unary naturals are not intended for computation, but rather for capturing structure, since it is relatively easy to reason about natural number arithmetic (by induction). There are also primitive numeric types (Int, Float, etc), but we could define a binary number type indexed over the corresponding $\mathbb{N}$ as follows:

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Bit} : \mathbb{N} \to \star\ \textbf{where} \\
\quad \begin{array}{rcl}
\mathsf{I} & : & \mathsf{Bit}\ (\mathsf{S}\ \mathsf{Z}) \\
\mathsf{O} & : & \mathsf{Bit}\ \mathsf{Z}
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Bin} : \mathbb{N} \to \mathbb{N} \to \star\ \textbf{where} \\
\quad \begin{array}{rcl}
\mathsf{Zero} & : & \mathsf{Bin}\ \mathsf{Z}\ \mathsf{Z} \\
(\#) & : & \mathsf{Bin}\ w\ n \to \mathsf{Bit}\ b \to \mathsf{Bin}\ (\mathsf{S}\ w)\ (b + n + n)
\end{array}
\end{array}
$$

Here, Bit x is a 1-bit number indexed by a value x which can be either zero or one, and Bin w n is a w-bit number indexed by its width in bits and the value it represents, n. An add-with-carry function for binary numbers would have the following type:

$$
\mathsf{add} : \mathsf{Bit}\ c \to \mathsf{Bin}\ w\ p \to \mathsf{Bin}\ w\ q \to \mathsf{Bin}\ (\mathsf{S}\ w)\ (c + p + q)
$$

Furthermore, any implementation would also respect the rules for unary addition, as explicitly stated in the type, and with a proof that Bin w n is unique for any given n, we can build proofs about add directly from proofs about $+$.

Unfortunately, $\#$ has *three* implicit $\mathbb{N}$ arguments:

$$
\begin{array}{rcl}
(\#) & : & \{w : \mathbb{N}\} \to \{n : \mathbb{N}\} \to \{b : \mathbb{N}\} \\
 & & \quad \to \mathsf{Bin}\ w\ n \to \mathsf{Bit}\ b \to \mathsf{Bin}\ (\mathsf{S}\ w)\ (b + n + n)
\end{array}
$$

In a naïve implementation, therefore, the memory footprint of a binary number is *exponential* with respect to the number of bits and the function add will also implicitly carry out the corresponding unary addition, since the unary number index is required for the type!

2.3 Possible solutions

There are two obvious possible solutions to this problem, and some less obvious possibilities. Two things we might try are:

$$D \quad ::= \quad \mathsf{name}\ x_0\ x_1\ \ldots\ x_n = T \tag{1}$$

$$\begin{array}{rcl} T & ::= & \mathrm{c} \mid v \mid \mathsf{name} \\ & \mid & T\ T \mid \lambda v : T.\ T \mid \Pi v : T.\ T \\ & \mid & \mathbf{let}\ v : T = T\ \mathbf{in}\ T \\ & \mid & \left\{ \begin{array}{l} \mathbf{case}\ T\ \mathbf{of} \\ \quad C\ v_0^0\ v_0^1\ \ldots\ v_0^{n_0}\ \Rightarrow\ T \\ \quad C\ v_1^0\ v_1^1\ \ldots\ v_1^{n_1}\ \Rightarrow\ T \\ \qquad \vdots \\ \quad C\ v_k^0\ v_k^1\ \ldots\ v_k^{n_k}\ \Rightarrow\ T \end{array} \right. \end{array} \tag{2}$$

Figure 1. Term syntax of the core calculus $\mathsf{TT}_{\mathsf{case}}$.

1. Erase all *implicit* arguments, and make it a compile time error to try to access them.
2. Erase all arguments which appear as *indices*, and make it a compile time error to try to access them.

To some extent, these approaches would allow us to preserve a clear phase distinction, similar to the separation between types and kinds in Haskell. Unfortunately, they would also limit the expressivity of the language. In practice, we often *want* run-time access to implicit arguments, leaving them implicit because they can be inferred by the machine, but using them for more efficient run-time computation. We believe this should be the programmer's choice, not the compiler's. For example, we can project the length out of a Vect directly by bringing the implicit n into scope:

$$\begin{array}{l} \mathsf{length} : \mathsf{Vect}\ n\ a \to \mathbb{N} \\ \mathsf{length}\ \{n\}\ xs = n \end{array}$$

The *forcing* optimisation [4] erases arguments from data structures which can be determined from their indices. This is also limited, however, in that it can only erase arguments which are guarded by constructors hence is insufficient for any of the RLE, U or Bin structures.

Agda and Coq each use programmer annotations (irrelevance annotations in the former, a separate universe Prop for proofs in the latter) and we would like to avoid this, because they limit the expressive power of the language. What's worse, both approaches are good at erasing *proofs* but they are not useful for erasing *indices*, which is what we need to do here.

A more promising approach, which we take as a starting point, is whole-program erasure analysis. We develop a method that coincides with existing approaches [16] in certain cases but also extends beyond them, namely in support of data structures with pattern matching and case analysis and we put it into practice for a complete dependently typed programming language.

3. Elaboration

Human-writable Idris programs are elaborated into the computer-friendly core calculus TT. Elaborated programs contain all information spelled out in detail so that they are machine-checkable and usable for further processing in the compiler.

3.1 The core calculi $\mathsf{TT}_{\mathsf{case}}$ and $\mathsf{IR}_{(\Box)}$

The core type theory of Idris, TT, allows function bodies only in the form of a single case tree with terms in the leaves. Case-expressions can inspect only variables and terms cannot contain case-expressions — these must be lifted out to separate functions if they occur in user code. In this paper, we describe a more general approach for a core type theory $\mathsf{TT}_{\mathsf{case}}$. In $\mathsf{TT}_{\mathsf{case}}$, an elaborated program consists of a collection of function definitions and data constructor declarations; the syntax of a single definition is shown in Equation (1) in Figure 1. The variables x_i stand for formal parameters of the function named name and its body is a term of the core type theory $\mathsf{TT}_{\mathsf{case}}$.

$$\begin{array}{rcl} T & ::= & \mathrm{c} \mid v \mid \mathsf{name} \\ & \mid & \Box \quad \textit{— only in } \mathsf{IR}_{\Box} \\ & \mid & T\ T \mid \lambda v.\ T \\ & \mid & \mathbf{let}\ v = T\ \mathbf{in}\ T \\ & \mid & \left\{ \begin{array}{l} \mathbf{case}\ T\ \mathbf{of} \\ \quad C\ v_0^0\ v_0^1\ \ldots\ v_0^{n_0}\ \Rightarrow\ T \\ \quad C\ v_1^0\ v_1^1\ \ldots\ v_1^{n_1}\ \Rightarrow\ T \\ \qquad \vdots \\ \quad C\ v_k^0\ v_k^1\ \ldots\ v_k^{n_k}\ \Rightarrow\ T \end{array} \right. \end{array}$$

Figure 2. Term syntax of the intermediate representations $\mathsf{IR}_{(\Box)}$. Our analysis and erasure works with this language.

Equation (2) shows that a $\mathsf{TT}_{\mathsf{case}}$ term is either a constant, a variable, a global name; an application, a lambda or a pi; a let-binding; or a case-expression, whose individual branches refer to data constructors C with their fields being pattern-matched as v_j^i.

After typechecking (and other related checks), the program is translated to the intermediate representation, where the shape of function definitions is the same, except that their bodies are expressed as terms of the intermediate language IR, shown in Figure 2.

The language IR differs from $\mathsf{TT}_{\mathsf{case}}$ mainly in being untyped; while individual values certainly do have types, these are no longer indicated in the syntax of the language, or checked again. This allows us to remove parts of the program arbitrarily, without worries about any typing constraints that would be invalidated by removing parts of the program.

Pruning takes a program in IR and produces a program expressed in $\mathsf{IR}_{\Box}$, a language identical to IR, except for an added symbol $\Box$, which represents an erased value. These are typically arguments to functions which are guaranteed never to be needed at run-time. Arities of functions are preserved; pruning may however reduce arities of data constructors, by removing any arguments which are guaranteed never to be accessed at run-time.

4. The Erasure Algorithm

Given an Idris program in the intermediate representation, our erasure algorithm aims to erase as much computation from it as possible, without changing the meaning of the program. First, we calculate *what* we can erase by means of usage analysis, then we *perform* the erasure by pruning the program.

4.1 Overview

We do not require any erasure annotations from the user, although they may choose to do so (see Section 4.7). Instead, we automatically infer what to erase, as far as possible. We say "as far as possible" because we do not necessarily find the most optimal solution due to the halting problem. Hence we informally define *unused* as "provably not necessary for runtime" and *used* as "not unused". This will be made more precise in Section 5.1.

We exploit the observation that functions *use* only some of their arguments (i.e. either by case-analysis or passing into usage-creating primitive functions.) Likewise, some fields of data constructors are never inspected. If we run usage analysis to find out what is

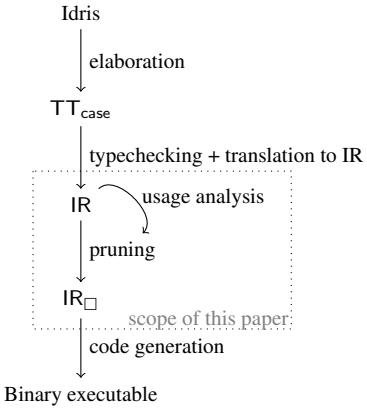


Figure 3. The big picture

guaranteed never to be be inspected at runtime, we can erase those pieces (arguments of functions and fields of data constructors) in the pruning phase.

The whole erasure process happens as pictured in Figure 3. First, usage analysis traverses programs expressed in IR and calculates which parts of the program may be necessary for runtime and which may be safely erased. Then this information enters the pruning procedure from IR to IR$_\square$.

4.2 Basic notions

The algorithm analyses definitions of functions. Starting from main, it searches the call graph to discover which functions are used in the program. For every function definition it builds a set of implications describing argument usage. The set of implications collected from the whole program is later solved to obtain the minimal consistent usage pattern.

Let us introduce a propositional variable for every *node*. A node is a pair containing a name of a function or a data constructor, and a natural number, which denotes the index of an argument of the name. Together, the two components of the pair identify a single item that can be erased or not. For the argument number i of the function f, let us call the corresponding variable f_i, and use $f_\star$ to denote the return value of f (which also may or may not be used). In the following text, we will use the name f_i for both the propositional variable and the node (function or constructor argument) itself.

The meaning of these propositional variables is: If and only if f_i is true, then the function f uses its i-th argument (if f is a function), or the data constructor f needs to store its i-th argument (if f is a data constructor).

We will proceed to build a set of implications. For this purpose, we only need implications in the form of Horn clauses, i.e.:

$$f_i \longleftarrow g_j, h_k, m_l, \ldots$$

The above implication says that if nodes g_j, h_k, and m_l, etc, are all found to be used, the node f_i has to be considered to be used as well. In the implementation, we additionally store a *reason* with every implication for purposes of error reporting, as described in Section 4.7.2. Let us summarise the terminology:

Global name A name of a function, data constructor, or other globally defined item, as opposed to e.g. lambda-bound names, which are *not* global names.

Node A pair (name, i) of a global name and argument number, as described above. A node stands for a position that can be considered erased or not.

$$[\![f\ x_0\ x_1\ \ldots\ x_n = B_f]\!] \quad = \quad [\![B_f]\!]^{\{x_0 \mapsto \{f_0\}, \ldots, x_n \mapsto \{f_n\}\}}_{\{f_\star\}} \tag{3}$$

$$\begin{array}{lcll} [\![\mathsf{c}]\!]^\Gamma_G & = & \emptyset & \textit{— constant} \\ [\![v]\!]^\Gamma_G & = & \{n_i \longleftarrow G \mid n_i \in \Gamma_v \setminus G\} & \textit{— variable} \\ [\![\mathsf{name}]\!]^\Gamma_G & = & \{\mathsf{name}_\star \longleftarrow G\} & \textit{— global} \end{array} \tag{4}$$

$$\begin{array}{l} [\![\mathsf{name}\ T_0\ T_1\ \ldots\ T_n]\!]^\Gamma_G \\ \quad = \{\mathsf{name}_\star \longleftarrow G\} \cup \bigcup_{i=0}^n [\![T_i]\!]^\Gamma_{\{\mathsf{name}_i\} \cup G} \end{array} \tag{5}$$

$$[\![F\ T_0\ T_1\ \ldots\ T_n]\!]^\Gamma_G \quad = \quad [\![F]\!]^\Gamma_G \cup \bigcup_{i=0}^n [\![T_i]\!]^\Gamma_G \tag{6}$$

$$\begin{array}{lcl} [\![\mathsf{let}\ x = T\ \mathsf{in}\ M]\!]^\Gamma_G & = & [\![T]\!]^\Gamma_G \cup [\![M]\!]^{\Gamma \cup \{x \mapsto \emptyset\}}_G \\ [\![\lambda x.\ M]\!]^\Gamma_G & = & [\![M]\!]^{\Gamma \cup \{x \mapsto \emptyset\}}_G \end{array} \tag{7}$$

Figure 4. Implication gathering: ordinary terms

Set of dependencies A set of nodes, usually associated with a variable. When that variable is recognised as used, it follows that all nodes from the set must be considered used as well.

Guards A set of nodes whose usage is prerequisite for recognising something else as used.

Implication A formula of the form $n_i \longleftarrow G$. Its meaning is "if every guard (node) from G is recognised as used, then n_i must also be considered used."

4.3 Implication gathering

As mentioned in Section 4.2, implication gathering combines a simple search of the call graph with inductive analysis of definitions of the functions encountered.

We describe implication gathering from a definition of a single function by defining the operation $[\![-]\!]^\Gamma_G$, which takes a term and returns a set of implications. The indices G and Γ are contexts: G being a set of *guards*; Γ being an environment that maps variable names to their dependency sets. We will explain these as we go.

Figure 4 and Figure 5 summarise the implication gathering process for ordinary terms (i.e. variables, applications, and bindings) and case expressions, respectively.

4.3.1 Function definitions

First, we η-expand function definitions, if applicable, generating fresh names for the new arguments. The rule is that any name defined with a functional type should have the corresponding number of binders in the outermost layers in its definition.

For example, instead of the function $\mathsf{makeEven} : \mathbb{N} \to \mathbb{N}$, defined as $\mathsf{makeEven} = \mathsf{double} \circ \mathsf{halve}$, we analyse the definition $\mathsf{makeEven}\ x = (\mathsf{double} \circ \mathsf{halve})\ x$, where x is a fresh variable.

The context Γ in $[\![-]\!]^\Gamma_G$ is used to keep track of every variable occurring in the function being analysed and to remember which dependencies it draws in, i.e. which nodes should be marked as used if this variable is found to be used. We call this set of nodes the *dependency set* of the variable, and thus the context Γ consists of assignments of the form $x \mapsto \{n_i, m_j, \ldots\}$, where the dependency set $\{n_i, m_j, \ldots\}$ is assigned to the variable x. We write Γ_x to denote the set of dependencies (nodes) assigned to x.

$$\left[\!\left[\begin{array}{l} \mathsf{case}\ T\ \mathsf{of} \\ \quad C_0\ y_0^0\ y_0^1\ \ldots\ y_0^{n_0}\ \Rightarrow\ B_0 \\ \quad C_1\ y_1^0\ y_1^1\ \ldots\ y_1^{n_1}\ \Rightarrow\ B_1 \\ \qquad\qquad \vdots \\ \quad C_k\ y_k^0\ y_k^1\ \ldots\ y_k^{n_k}\ \Rightarrow\ B_k \end{array} \right]\!\right]_G^\Gamma = [\![T]\!]_G^\Gamma \cup \bigcup_{i=0}^{k} [\![B_i]\!]_G^{\Gamma \cup V_i} \qquad V_i = \left\{ y_i^j \mapsto \{(C_i)_j\} \;\middle|\; j \in 0 \ldots n_i \right\} \tag{8}$$

$$\left[\!\left[\begin{array}{l} \mathsf{case}\ x\ \mathsf{of} \\ \quad C\ y^0\ y^1\ \ldots\ y^n\ \Rightarrow\ B \end{array} \right]\!\right]_G^\Gamma = [\![B]\!]_G^{\Gamma \cup V} \qquad V = \left\{ y^j \mapsto \{C_j\} \cup \Gamma_x \;\middle|\; j \in 0 \ldots n_i \right\} \tag{9}$$

Figure 5. Implication gathering: case expressions

For example, the variable x in the above function makeEven *is* the first argument of the function so its usage should mark $\mathsf{makeEven}_0$ as used: $(x \mapsto \{\mathsf{makeEven}_0\}) \in \Gamma$. A variable may draw in a non-singleton set of dependencies if it arises from nested pattern matching.

Hence, we start by defining the set of implications for a single function definition, as shown in Equation (3) in Figure 4. In the equation, f is the name of the function being defined, $x_0 \ldots x_n$ are its formal arguments, and B_f is its body. We start the analysis of the body by defining $G = \{f_\star\}$, which says that every implication arising from this function will contain $f_\star$, the node representing the *return value* of f, among its assumptions. We also define $\Gamma = \{x_0 \mapsto \{f_0\}, \ldots, x_n \mapsto \{f_n\}\}$, which assigns the appropriate dependency set $\{f_i\}$ to every formal argument x_i.

4.3.2 Terms

Equation (4) shows how to build implication sets for constants, local variables and global names. In an implementation, for instance our implementation in Idris, there may be other cases to consider, such as de Bruijn indexed variables. We omit these additional cases in this description but, for example, with de Bruijn indices the idea is that we keep an additional context: a *stack* of dependency sets. (Recall that Γ is a *mapping* from names to dependency sets.)

4.3.3 Applications

Applications are central to our notion of erasure. If we view functions/operators as tree nodes with operands for children, then erasure corresponds to pruning the tree. For example, consider when the variable x is used in the following piece of code:

$$f\ x = g\ y\ (h\ x)\ z$$

If g does not use its second argument, the whole expression $(h\ x)$ can be discarded to save computation, which means that x is not used here. Even if g_1 is used, it may be the case that h does not use its first argument, which would make x unused again. In other cases however, we have to mark x as used.

As we have seen, the usage of x depends on g_1 and h_0, and every occurrence of every variable has got such a set of "preconditions" derived from its enclosing environment, from the path down the tree of applications. We call this set of preconditions *guards* and maintain it as the index G in $[\![-]\!]_G^\Gamma$.

We can generalise the above example and write a general rule for application of global names (global functions or data constructors), as shown in Equation (5). In this case, for every actual argument T_i, we include its set of implications, guarded by f_i; this condition added to G expresses that these implications come into effect only if the i-th argument of f is used.

If the inspected term is an application of anything other than a global name, such as a local variable or a more complex expression, we assume nothing is erased. In practice, the only difference between applying a global name and applying anything else is that we extend the set of guards G in the operands when applying global names.

Also, since Idris uses strict evaluation, we do not perform deep analysis to find whether arguments of non-global applications are used, but consider them used because they will be evaluated regardless. This is a simplification; we could extend our erasure to erase from internal lambdas and let-bindings in future work. This decision has not proved to be problematic in practice because explicitly mentioned data is usually not intended to be erased.

Therefore, in cases other than application of a global name (such as application of a local variable or a lambda) we include the dependencies for the operator and all operands, as shown in Equation (6).

4.3.4 Lambdas and let-bindings

Lambdas and let-bindings are analysed as described in Equation (7). We opt for the easy strict approach here, as mentioned above: all lambda bindings are considered used at the point of application and all let-bound values are considered used at the point of let-binding. This allows us to leave the dependency sets Γ_x of let-bound and lambda-bound variables empty because usage of the bound variables does not affect what needs to be evaluated.

4.3.5 Case expressions

Analysis of the most general case expression is shown in Equation (8) in Figure 5. What contributes to its set of implications is the inspected term T and each branch of the case expression.

Notice that in each branch, the context Γ is extended with V_i, which describes the new variables introduced by pattern matching. V_i expresses the fact that the value y_i^j is obtained by reading the j-th field of the constructor C_i, represented by the node $(C_i)_j$.

In Equation (9), we include one special case of case-expressions: when the case expression inspects a *variable* with only one possible branch, we defer usage of the inspected variable to the point of usage of the projected variables. The reason is that if all projections end up in erased positions or not used at all (for illustration, this usually happens when matching on refl), we want to keep the inspected variable marked as unused as well. In such cases, the compiler *must* prune unused case-matches in order to keep the inspected expression unevaluated.

We could do the same for singleton case-expressions inspecting arbitrary *terms* but that would require a more elaborate formulation of dependency sets than we currently have. Therefore, these terms will not be erased. This is sufficient for Idris, since after elaboration case-expressions inspect variables, never compound terms.

4.4 Special cases

In a complete language implementation, there are additional language elements that interact with erasure. For Idris this includes primitive types, compiler builtin functions, and type classes.

4.4.1 Primitives and builtins

The above erasure mechanism generates implications only from function definitions. However, primitives, foreign calls and compiler builtins have no in-language definitions and they would therefore end up recognised as not using anything.

For that reason, we introduce *usage postulates*, which assert which arguments of which builtins are used. In the case of Idris there are usage postulates built into the compiler, and they can be introduced by a programmer with a compiler pragma.

The most important usage postulate states that the entry point of the program is the function main:

$$\mathsf{main}_\star \longleftarrow \emptyset \tag{10}$$

4.4.2 Type classes

Our erasure scheme cannot yet express higher-order erasure patterns, such as data constructors taking functions with erased arguments, or functions taking functions with erased arguments. One important instance of this pattern is type classes. In Idris, these are implemented as records containing (usually) functional fields. However, we would like to erase from methods of type classes in the same way as from normal functions.

The approach we take is to create special nodes for each argument of each method of each type class. Class methods have no definition; the only functions available come from the instances. We define the usage pattern of a method as the union of usage of all its instances mentioned in the program.

For example, we define the usage of $+$ as the union of the usage of $+_{\mathbb{N}}$, $+_{\mathsf{Int}}$, $+_{\mathsf{Double}}$, etc. More precisely, we define the usage of a method as the union of the usage of whatever is ever passed to the instance constructor. In Idris, this may include runtime-constructed instances.

4.5 Dependency solving

The output of implication gathering after processing all function definitions from the set $\mathcal{F}$ of functions in the program, including all postulates or extra implications from Section 4.4, is a set of implications $\mathcal{O}$, where every implication has the form $n_i \longleftarrow G$:

$$\mathcal{O} \stackrel{\text{def}}{\equiv} \mathsf{Postulates} \cup \bigcup_{f \in \mathcal{F}} [\![f]\!] \tag{11}$$

Recall from Section 4.2 that we identify every node with a propositional variable of the same name. Hence $\mathcal{O}$ is also a logic program, from which we can tell which propositional variables *must* be true (which arguments of which functions are used and thus cannot be erased.) We can then declare all other nodes unused and perform the corresponding erasure.

Therefore, we run forward chaining on the set of implications $\mathcal{O}$; the output of this step is the set of used nodes, which we call $\mathcal{U}$. Formally, $\mathcal{U}$ is the minimal set such that:

$$G \subseteq \mathcal{U} \Rightarrow n_i \in \mathcal{U} \quad \text{for all} \quad (n_i \longleftarrow G) \in \mathcal{O} \tag{12}$$

4.6 Erasure

The actual removal of irrelevant computation happens by pruning the intermediate representation of programs, transforming a program expressed in IR to a program expressed in $\mathsf{IR}_\square$. We represent pruning by the operation $\lfloor - \rfloor_{\mathcal{U}}$, defined in Figure 6.

4.6.1 Simple terms

Equation (13) shows how to prune simple terms, such as constants, variables, global names, applications, lambdas and let-expressions. These do not change beyond being pruned recursively.

There are two exceptions to the application rule: applications of globally named functions are pruned according to Equation (14)

$$\begin{array}{lcl}
\lfloor \mathrm{c} \rfloor_{\mathcal{U}} & \stackrel{\text{def}}{\equiv} & \mathrm{c} \\
\lfloor v \rfloor_{\mathcal{U}} & \stackrel{\text{def}}{\equiv} & v \\
\lfloor \mathsf{n} \rfloor_{\mathcal{U}} & \stackrel{\text{def}}{\equiv} & \mathsf{n} \\
\lfloor \lambda v.\ T \rfloor_{\mathcal{U}} & \stackrel{\text{def}}{\equiv} & \lambda v.\ \lfloor T \rfloor_{\mathcal{U}} \\
\lfloor \textbf{let } v = T \textbf{ in } M \rfloor_{\mathcal{U}} & \stackrel{\text{def}}{\equiv} & \textbf{let } v = \lfloor T \rfloor_{\mathcal{U}} \textbf{ in } \lfloor M \rfloor_{\mathcal{U}} \\
\lfloor F\ T_0 \ldots T_n \rfloor_{\mathcal{U}} & \stackrel{\text{def}}{\equiv} & \lfloor F \rfloor_{\mathcal{U}}\ \lfloor T_0 \rfloor_{\mathcal{U}} \ldots \lfloor T_n \rfloor_{\mathcal{U}} \\
\multicolumn{3}{l}{\textbf{where}} \\
\multicolumn{3}{l}{\quad F \text{ is not a global name}}
\end{array} \tag{13}$$

$$\begin{array}{l}
\lfloor \mathsf{name}\ T_0\ T_1\ \ldots\ T_n \rfloor_{\mathcal{U}} \stackrel{\text{def}}{\equiv} \mathsf{name}\ \lfloor X_0 \rfloor_{\mathcal{U}}\ \ldots\ \lfloor X_n \rfloor_{\mathcal{U}} \\
\textbf{where} \\
\quad X_i \stackrel{\text{def}}{\equiv} \begin{cases} \square & \text{if } (\mathsf{name}_i \notin \mathcal{U}) \\ & \quad \wedge \big(i < \mathsf{arity}(\mathsf{name})\big) \\ T_i & \text{otherwise} \end{cases}
\end{array} \tag{14}$$

$$\begin{array}{l}
\lfloor C\ T_0\ T_1\ \ldots\ T_n \rfloor_{\mathcal{U}} \stackrel{\text{def}}{\equiv} \lfloor C \rfloor_{\mathcal{U}}\ \lfloor T_{a_0} \rfloor_{\mathcal{U}}\ \ldots\ \lfloor T_{a_k} \rfloor_{\mathcal{U}} \\
\textbf{where} \\
\quad \{a_i\}_{i=0}^{k} \stackrel{\text{def}}{\equiv} \text{maximal subsequence of } 0 \ldots n \\
\qquad\qquad \text{where } \forall i.\ C_{a_i} \in \mathcal{U} \\
\quad \lfloor C \rfloor_{\mathcal{U}} \stackrel{\text{def}}{\equiv} \text{a reduced variant of } C \\
\qquad\qquad \text{restricted to } k+1 \text{ fields}
\end{array} \tag{15}$$

$$\begin{array}{l}
\left\lfloor \begin{array}{l} \textbf{case } T \textbf{ of} \\ \quad C_0\ y_0^0\ y_0^1\ \ldots\ y_0^{n_0}\ \Rightarrow\ B_0 \\ \quad C_1\ y_1^0\ y_1^1\ \ldots\ y_1^{n_1}\ \Rightarrow\ B_1 \\ \qquad\qquad \vdots \\ \quad C_k\ y_k^0\ y_k^1\ \ldots\ y_k^{n_k}\ \Rightarrow\ B_k \end{array} \right\rfloor_{\mathcal{U}} \stackrel{\text{def}}{\equiv} \\
\qquad \stackrel{\text{def}}{\equiv} \left\{ \begin{array}{l} \textbf{case } \lfloor T \rfloor_{\mathcal{U}} \textbf{ of} \\ \quad \left\lfloor C_0\ y_0^0\ y_0^1\ \ldots\ y_0^{n_0} \right\rfloor_{\mathcal{U}}\ \Rightarrow\ \lfloor B_0 \rfloor_{\mathcal{U}} \\ \quad \left\lfloor C_1\ y_1^0\ y_1^1\ \ldots\ y_1^{n_1} \right\rfloor_{\mathcal{U}}\ \Rightarrow\ \lfloor B_1 \rfloor_{\mathcal{U}} \\ \qquad\qquad \vdots \\ \quad \left\lfloor C_k\ y_k^0\ y_k^1\ \ldots\ y_k^{n_k} \right\rfloor_{\mathcal{U}}\ \Rightarrow\ \lfloor B_k \rfloor_{\mathcal{U}} \end{array} \right.
\end{array} \tag{16}$$

$$\begin{array}{l}
\left\lfloor \begin{array}{l} \textbf{case } x \textbf{ of} \\ \quad C\ y^0\ y^1\ \ldots\ y^n\ \Rightarrow\ B \end{array} \right\rfloor_{\mathcal{U}} \stackrel{\text{def}}{\equiv} \lfloor B \rfloor_{\mathcal{U}} \\
\textbf{if} \\
\quad \forall_{i=0}^{n} (y^i \text{ not free in } \lfloor B \rfloor_{\mathcal{U}})
\end{array} \tag{17}$$

Figure 6. Pruning the intermediate representation

and applications of data constructors are pruned according to Equation (15), which we elaborate in the following two sections.

4.6.2 Functions with global names

We prune the computation tree by replacing erased positions with a placeholder $\square$ that will compile to a "null" value in the final stages of compiling, as shown in Equation (14). We do not change function arities; instead, we do not perform the pruned pieces of computation and leave the corresponding references undefined in the low-level representation.

```
data Vect : ℕ → ⋆ → ⋆ where
  []   :  Vect Z a
  (::) :  .{n : ℕ} → (x : a)                    (18)
          → (xs : Vect n a)
          → Vect (S n) a

index : .{n : ℕ} → Fin n → Vect n a → a
index    FZ     (x :: xs)  =  x
index  (FS n)   (x :: xs)  =  index n xs
```

Figure 7. Syntax of erasure annotations in Idris

If an application is not saturated, it may need η-expansion or other changes depending on the context to ensure that all arguments end up in the correct positions.

4.6.3 Data constructors

Data constructor applications are simpler to handle than functions because they have clearly defined arities, and other parts of the compiler keep them exactly saturated in the internal representation. Therefore, we completely *remove* their fields and change their arities, instead of filling them with placeholders, as shown in Equation (15).

4.6.4 Case-expressions

Case-expressions are pruned as described in Equation (16). We recursively prune the inspected term and all right-hand sides. However, we must also prune the left-hand sides using the rule described in Equation (15) because the data constructors' runtime arities might be different.

If we used the singleton-case optimisation from Section 4.3.5 specifically, Equation (9), we must also prune the singleton case-trees according to Equation (17) to ensure that x is not deconstructed at runtime as the corresponding low-level reference would be undefined.

The resulting pruned program is now free from the runtime-irrelevant data we have identified, and is ready to be compiled further by the back-end to an executable.

4.7 Erasure annotations

We also allow user-provided erasure annotations. These serve two purposes: requesting warnings if a certain part of the program is *not* erased, and influencing case-tree elaboration.

In Idris, users can assert that certain arguments of functions and data constructors will be found unused by the usage analysis by putting dots in the corresponding places of type signatures. In Figure 7, this is demonstrated on the type index n, in both the data constructor $(::)$ and the function index.

If this assertion fails because usage analysis finds that n could be used, the compiler will emit a warning and will not erase n, still producing a correct program.

It is important to note that erasure does not depend on erasure annotations, although they can improve it, and the erasure mechanism will typically erase much more than the user has annotated.

The subsequent compilation stages perform an independent usage analysis. If this marks something as erased, there is no reason to leave it in the program even if the user did not annotate it. Conversely, when analysis detects that the runtime code refers to a value, it *cannot* be erased, even if marked as such.

Furthermore, Idris allows implicit quantification of unbound variables. In Figure 7, the parameter a of the data constructor $(::)$ is an *unbound implicit* variable, while n is a *bound implicit*. For unbound implicits, Idris will infer their types and quantify them implicitly, but it will also consider them dotted, i.e. marked for erasure. Indeed, we included the $.\{n : \mathbb{N}\}$ part purely for illustrative purposes; we could have left n implicitly quantified, which would also make it dotted, yielding the exact same result.

Since the core type theory has no notion of erasure annotations, type signatures with erasure annotations are elaborated ignoring those annotations completely. Erasure annotations are stored in a separate structure.

4.7.1 Case-tree elaboration

When elaborating pattern-matching function definitions into case trees, there may be several choices. Consider the following:

```
halve : (n : ℕ) → Even n → ℕ
halve      Z          EvenZ       =  Z
halve  (S (S n))  (EvenSS e)      =  S (halve n e)
```

The run-time version of the program could case-split on either the first or the second argument; both would work *correctly* but the two options have different erasure properties, and thus different runtime behaviour, possibly with even asymptotically different runtimes and/or memory usage.

In these cases, the user can influence the case-tree elaborator by putting an explicit erasure annotation in the type of the function and the elaborator will try to avoid using the dotted arguments, preferring other solutions, if available.

The Idris compiler uses a variant of the pattern compiling algorithm described by Wadler [21]. Besides other modifications and heuristics, the algorithm has been modified to reorder the list of pattern variables so that no dotted variables (including those that arise from matching on constructors) are matched before exhausting non-dotted variables.

4.7.2 Erasure warnings

After usage analysis, the compiler checks whether everything that has been dotted (marked with erasure annotations) is also recognised as unused. If that is not the case, it reports all cases where an erasure annotation is violated.

As already briefly mentioned, each implication gathered from a program also carries a *reason*, which is a piece of data that describes how the implication arose. After solving the resulting logic program, each reachable node also carries a set of reasons from the implications that contributed to its usage. We include this information in warnings to make them usable in debugging.

Another thing to note is that erasure warnings are merely warnings saying that the programmer's idea differs from what has been found about the source code. The subsequent compilation stages use *only* the output of usage analysis (not the annotations) and the resulting program *will* work correctly even with erasure violations. However, it may be less efficient than the programmer expected.

5. Soundness

In this section, we provide a justification of the soundness of the erasure algorithm. Note that soundness of erasure corresponds to completeness of usage analysis (and vice versa) and for that reason it is necessary to always specify which one we are talking about. Also note that it is impossible to have an algorithm that is both sound and complete, because usage of a variable may depend on the outcome of a function, which is itself undecidable. Therefore we err on the sound side.

5.1 Data flow graph

The algorithm as we present it in Section 4 is fairly efficient and simple but reasoning about it turns out to be a difficult problem. The reason is its non-locality: erasure at any single place depends on

the behaviour of the whole program and thus it is not amenable to inductive reasoning.

Therefore, it is useful to recognise that the algorithm is actually a special case of a more general notion, which, besides making reasoning feasible, will also help us to pinpoint data constructors as the source of the non-locality.

In Section 4.2, we introduced the notion of *nodes*. We can intuitively view nodes as vertices of a data flow graph and use edges to model data dependencies between nodes.

Let us therefore define a (directed) data dependency graph G for a program. The set of vertices V will contain nodes $f_\star$, and $f_0, f_1, \ldots, f_{n-1}$ for every globally named function or data constructor of arity n.

The arguments are represented by the nodes $f_0, \ldots, f_{n-1}$ and the return value is represented by $f_\star$. However, in order to define the edges of the graph, we need to make it clear what it *means* for a node to depend on another node. We also introduce the node $C_\star$ for every data constructor C, which we omitted in Section 4.

Aside: Dependently typed functions, such as printf, may be variadic. We took the liberty of defining the arity as the number of formal arguments in the text of the definition of a function, which works well in practice.

Definition 1 (Node instances)**.** An *instance* of the node n_i is:

- if n is the name of a function — the variable standing for the i-th formal argument of n within the body of n;
- if n is the name of a data constructor — any variable that arose from pattern matching on the i-th field of n. Note that in $\mathsf{IR}_{(\Box)}$, there are no nested patterns, since these are represented as nested case-expressions, and every constructor field corresponds to exactly one variable.
- if $n_i = f_\star$ — any saturated application of f. The function or data constructor of arity n is applied to at least n arguments.

In practice, not all applications of constructors and globally named functions are saturated and η-expansion may be needed to convert them to the desired form.

Definition 2 (Substitution of instances)**.** If E is a term, $E[n_i := x]$ represents a variant of E where all instances of n_i have been replaced with x. This definition also extends to whole programs.

Definition 3 (Terms in modified programs)**.** As defined in Section 4.3.1, B_f stands for the body of the function. To study behaviour of modified programs, we define B_f^P to stand for the body of the function f in the program P. Then we can have $B_f^{P'}$ stand for the body of f in a modified program P'.

Definition 4 (Equivalence of terms)**.** Let $M \sim N$ mean that the terms M and N are not "observably distinguishable" in the calculus we are working in.

We leave this notion abstract because depending on the reduction behaviour of the calculus in question, this equivalence could take on different forms, ranging from literal equality, through equality of normal forms, to more complicated criteria. If it is too strict, too little will be erased; if it is too lax, erasure might not be sound.

The general guideline is that $\sim$ should be decidable and $M \sim N$ should imply that M and N "behave the same." This is the sense in which we require the erased program to "behave the same" as the original program.

Definition 5 (Usage patterns)**.**

- Let P be a program expressed in IR.
- Let N be the set of nodes in the program P, as defined in Section 4.2.
- A *usage pattern* is a set $U \subseteq N$.
- Let

$$\lfloor P \rfloor_U \stackrel{\text{def}}{\equiv} P[n_1 \ldots n_k := \Box], \quad \{n_1, \ldots, n_k\} = N \setminus U$$

  denote the program *erased* according to the usage pattern U.
- Usage pattern U is *consistent* if and only if erasure by U does not affect the reduction behaviour of the body of the function main:

$$B^P_{\mathsf{main}} \sim B^{\lfloor P \rfloor_U}_{\mathsf{main}}$$

Observation: A superset of a consistent usage pattern is consistent. This again, however, depends on the reduction behaviour of the language in question. $\mathsf{IR}_\Box$ of Idris allows passing undefined values $\Box$ in variables and function arguments freely, as long as they are not inspected.

Definition 6. Let $e'(T, n)$ denote that the term T *depends* on the node n:

$$e'(T, n) \stackrel{\text{def}}{\equiv} T[n := \Box] \not\sim T$$

Definition 7 (Edges of usage graph)**.** Now we can define what the edges of the usage graph are by defining the edge predicate e. Intuitively, there is an edge $e(m, n)$ if m "depends on" n. More precisely, the graph contains these edges:

- $e(f_\star, f_i)$ iff f is a function and $e'(B_f, f_i)$; i.e. the return value of the function depends on its i-th argument.
- $e(f_\star, g_\star)$ iff f is a function, g is either a function or a constructor, and $e'(B_f, g_\star)$; i.e. the return value of f depends on the return value of g.
- $e(f_\star, C_i)$ iff C is a data constructor and $e'(B_f, C_i)$; i.e. the return value of f is affected by a value read from the i-th field of the constructor C.
- $e(C_\star, C_i)$ iff $p(\mathsf{main}_\star, C_i)$; i.e. it matters what we save in the i-th field of the constructor C because reading from it somewhere affects main.

The last point refers to the predicate p, which we now define as the path predicate:

$$p(m, n) \stackrel{\text{def}}{\equiv} m = n \ \vee \ \exists x.\, e(m, x) \wedge p(x, n)$$

The odd one out among the edges is the case for $e(C_\star, C_i)$, which says that an application of a constructor depends on its i-th argument whenever the return value of main depends on data read from that field *anywhere in the program*. This is precisely the non-local dependency that data constructors introduce into the reasoning.

Definition 8 (Used nodes)**.** Let us define what it means for a node n to be used:

$$u(n) \stackrel{\text{def}}{\equiv} p(\mathsf{main}_\star, n)$$

Now we arrived at a precise (up to the notion of equivalence) explanation of what it means for a node to be used and we have also defined terminology and infrastructure to reason about usage. This also shows that our algorithm is in fact a reachability search starting from $\mathsf{main}_\star$, searching for all nodes that represent values possibly influencing the result of main.

5.2 Proof Sketch

We now express what it means for erasure to be sound and sketch the proof steps.

Claim 1. *Erasing the program P according to the result of our usage analysis does not change the behaviour of* main*. More precisely,*

$$B^{\lfloor P \rfloor_{\mathcal{U}}}_{main} \sim B^P_{main} \tag{19}$$

where the set $\mathcal{U}$ is the result of usage analysis, as described in Section 4.5.

Step 1 (Completeness of data flow presentation)**.** *If we cannot deduce that the term E depends on the node n, we can erase all instances of n in E without affecting its behaviour:*

$$\not\vdash p'(E, n) \Longrightarrow E^{P[n:=\Box]} \sim E$$

Step 2 (Consistency of u)**.** *The usage pattern $\{n \mid u(n)\}$ is consistent.*

Step 3. *The implication gathering algorithm in Section 4.3 finds all edges of the graph and combines them with the definition of the path predicate to generate reachability implications.*

The antecedent of every implication corresponds to a condition of existence of edge from Definition 7.

(If the algorithm finds more implications than there are edges in the graph, it is not a problem for soundness.)

Step 4. *The set $\mathcal{U}$ calculated by forward chaining, as described in Section 4.5, therefore contains $\{n \mid u(n)\}$.*

Step 5. *Hence, $\mathcal{U}$ is a consistent usage pattern and $B^{\lfloor P \rfloor_{\mathcal{U}}}_{main} \sim B^{P}_{main}$.*

6. Results

The presented algorithm is implemented in the Idris compiler, and by default runs on all programs. In this section, we show how it performs in practice for the motivating examples presented earlier.

6.1 Benchmarks and measurements

We ran three different programs,with and without erasure, on inputs of different size, 10 times per input, on a regular desktop computer. These benchmarks, and others, can be found online at `https://github.com/ziman/idris-benchmarks/`.

Figure 8(a) shows the runtime of the binary adder described in Section 2.2. We can see that the unerased variant slows down exponentially, while the runtime of the erased program grows only linearly.

Figure 8(b) shows the runtime of a palindrome decider shown in Section 2.1. The index xs in the constructor Two makes the whole structure quadratically large. In fact, linear regression in the log-log runtime graph shows that our implementation without erasure runs in cubic time. Again, after erasure, the runtime returns to linear growth, which is asymptotically optimal and almost not visible in the plot.

Finally, Figure 8(c) shows that erasure can help even if it does not improve asymptotic runtime of a program. This is why we chose this program as a motivating example: in our benchmark, we simply generate, compress and decompress a list, which in total takes linear time, even if the "compressed" form is (asymptotically) wasteful and actually *not* compressed. The program performs run-length compression and decompression of a list of a given length. The central data structure used is shown in Section 1 and is linear in size, thanks to sharing between its indices. Therefore, erasure does not yield asymptotic improvements, it simply eliminates unnecessary index manipulation.

Figure 8(d) also shows that erased programs take less time to compile. This may seem surprising at first, but there is typically a lot of irrelevant code in programs with many invariants expressed in types. While the erasure analysis takes time, there is significantly less code to generate.

7. Related Work

***Coq and* Prop** The Coq proof system has a whole universe of types designated for erasure, named Prop. Coq generates executable code via program extraction [18, 12, 13] and all values belonging to Prop are removed during the extraction. When defining a data type, a programmer decides whether the data type should belong to the universe of runtime-relevant values (Type) or to the universe of erasable proofs (Prop).

```
data Bin : .ℕ → ⋆ where
  N  :  Bin 0
  I  :  ∀n → Bin n → Bin (1 + n + n)
  O  :  ∀n → Bin n → Bin (0 + n + n)
```
(20)

```
fishy : Bin 0   — type says value represents 0
fishy = I 0 N   — in fact, value represents 1
```
(21)

Figure 9. A type-correct Agda program using irrelevant indices.

Unfortunately, it is not possible to erase indices, which usually have a non-Prop type, such as the list in our introductory example. We could duplicate the type of lists in Prop (and the related functions) but we still have to deal with cases like the following,

$$\mathsf{snocView} : (xs : \mathsf{List}\ a) \to \mathsf{SnocView}\ xs$$

where xs cannot be in Prop because execution depends on it.

Leivent shows [11] that with sufficient care, we can avoid manual duplication of the whole Type universe in Prop by defining a suitable embedding into Prop in the spirit of Letouzey and Spitters' nc monad [14] and this approach is practical enough to implement interesting programs (such as red-black trees) that extract to very clean ML code.

However, it is still not ideal for multiple reasons: it requires heavy automation and it's difficult to achieve good results even with Coq's automation facilities; it requires an extra axiom which contradicts proof irrelevance [10]; it was discovered that this formulation of erasability was inconsistent when not restricted to just Type_0 [6], and it is still unclear whether it is consistent with the restriction.

Agda and irrelevance In Agda, types are not inherently erased or otherwise, but any type can be made *irrelevant* by putting a dot in front of it [20]. Irrelevance has certain consequences but the important one for us is that irrelevant arguments are erased.

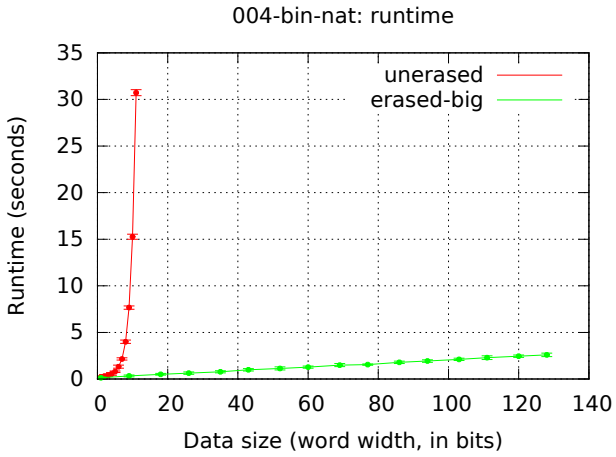
However, as in the case of Coq; while this is a good approach to erasing *proofs*, it is not suitable for erasing indices, as illustrated in the type-correct program shown in Figure 9.

This is because, besides other things, two values of an irrelevant type are considered equal. In other words, in Agda, irrelevant values are conflated *too early*: already at the point of typechecking. What we need instead is to have the typechecker treat the indices as entirely ordinary and distinguishable values and erase only afterwards.

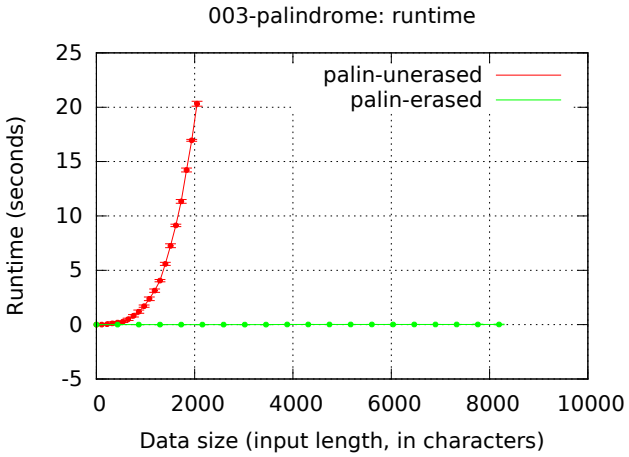
Forcing and collapsing Previously, Idris employed the forcing and collapsing optimisation [4]. This optimisation can erase some important classes of runtime-irrelevant data, such as termination/accessibility predicates [2] or those indices that are reconstructible from the data being indexed. Since all data erased by this optimisation must not be manipulated by the generated code, it is subsumed by our new approach, which recognises such cases from the generated code.

Erasure pure type systems In his dissertation [16], Mishra-Linger defines a whole erasure framework, together with an extensive metatheoretic analysis. His approach is to annotate every binder with a binary flag denoting the runtime-relevance of the bound variable. The dissertation also includes an annotation inference algorithm for pure type systems, which is provably optimal in terms of the count of annotations marked as runtime-irrelevant (and therefore erased).

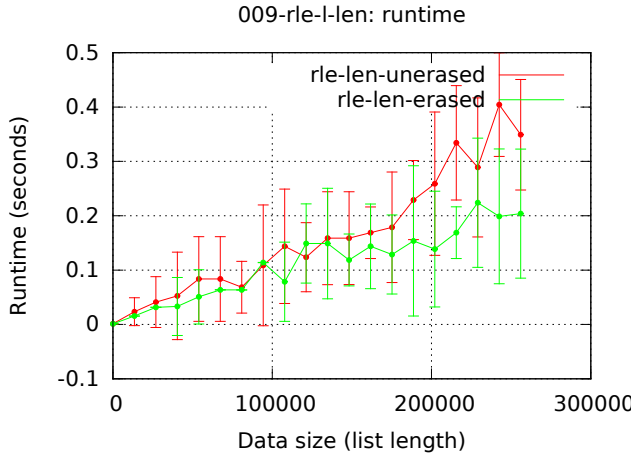
Despite being formulated differently, this approach to inference of annotations (collecting constraints in CNF and using unit propa-



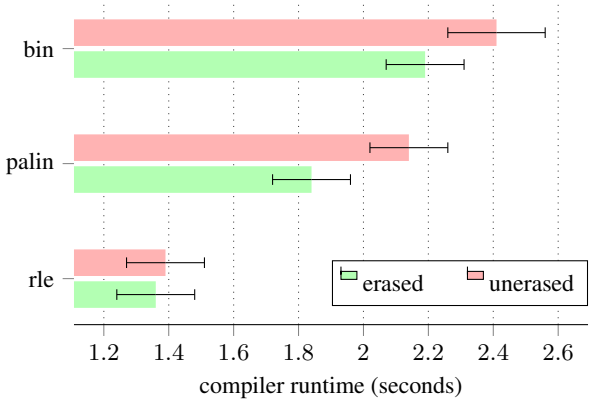
(a) Run time of the binary adder, $O(c^n)$ to $O(n)$



(b) Run time of the palindrome checker, $O(n^3)$ to $O(n)$



(c) A non-asymptotically improved program, RLE



(d) Compilation times

Figure 8. Compilation time and runtime impact of erasure. Error bars show $\pm 3\times$ sample standard deviation.

gation to solve them) coincides with ours in the cases where only the PTS subset of our calculus is used.

However, while it can erase from lambdas (unlike our approach currently), it does not address case-expressions or pattern matching on data constructors and the global constraints arising there. Instead, Mishra-Linger discusses eliminators, which are equivalent to pattern-matching [7] in theory. Our graph-theoretic presentation in Section 5 allows us to see a deeper structure behind the constraints, extend the analysis and reason about it. A language inspired by EPTS from this dissertation would, however, be a good target language for our erasure analysis.

Dependent Haskell A similar approach is presented by Gundry in his design of Dependent Haskell [8] and in a joint paper with McBride [9]. Starting with a simple type theory, every type judgement is annotated with some value from a preorder of phases and typability of a term at some phase implies its typability at any higher phase. The mechanism then provides an erasure procedure, which erases parts of terms typed at phases from the chosen boundary upwards.

Trellys The Trellys project [5] gave rise to several dependently typed languages, which in general share the property of having a restricted fragment used to express logic in an otherwise non-total and permissive language. Again, type judgements are annotated with relevance levels (L-ogic or P-rogrammatic) and these languages could potentially be used as a target language for our usage analysis.

Type Theory in Colour Jean-Philippe Bernardy and Guilhem Moulin present Type Theory in colour [1], namely a calculus called CCCC. In CCCC, every type judgement is annotated with a set of *colours* called *taint* and usage of tainted variables taints whole expressions. There is an erasure procedure that removes those parts of terms that are tainted by any selected colour. While CCCC could also be used as a target language of erasure inference, it is more general than that and erasure of dead data is only one of the aspects which may be modelled by colour.

Multi-stage programming Multi-stage programming [19] presents the phase distinction from a different perspective: some parts of the code are executed at compile time, some at run time. Unsurprisingly, the run-time portions can refer to the results of the compile-time parts of code but not vice versa. After compiling, analogously to erasure, the compile-time computation is not present in the resulting run-time code.

Laziness It might seem that lazy evaluation would relieve us from evaluating arguments that are never inspected. In fact, laziness as an erasure mechanism is effective only with highly compressible structures, such as the first 1000 Fibonacci numbers, or when the computation involved is expensive. Otherwise, we have to have the relevant data stored *somewhere*, even if the routine that processes it has not been run yet. It is not very helpful to store a big chain of thunks instead of a big list. Therefore, since we can do better and we can discover most of dead computation at compilation time, we prefer simply to avoid including it in the resulting program at all.

8. Discussion

Our approach to erasure has several advantages, but also a number of limitations at present. An important advantage is that introduction of this form of erasure does not require any changes to the type theory of the language in question. It can be regarded as an optimisation that spots dead code/data in the intermediate representation of programs and removes it, but does not affect typechecking at all. By consequence, introduction of this optimisation does not change the set of compilable/checkable programs, although it may issue new compiler warnings.

No user-provided annotations in the source code are required and most programs do not need any change. We did, however, change some Idris programs according to the warnings coming from usage analysis because they uncovered hidden inefficiencies.

Users *can* use annotations, for two purposes:

- to mark parts of programs as erased in order to get warnings in case they are not, which would mean that the program does not work as intended;
- to influence the case-tree elaborator to make a different choice if there are more ways to build a case tree from a set of patterns, as shown in Section 4.7.1.

Usage analysis is run after case-tree elaboration, which means that the case-tree elaborator cannot use results of the analysis. Instead, in Idris, the elaborator uses heuristics, unless user-provided annotations override them.

Idris features the newtype optimisation, which means representing a single-constructor, single-field data type as just the field itself. In Coq extraction, this is known as the (opposite of the) KeepSingleton extraction option. For example, if a record contains just a single Int field, the whole record will be represented as an Int at runtime. Since erasure removes fields from data constructors, it often makes them subject to the newtype optimisation. This combination of optimisations can be applied repeatedly and thus compile rich nested structures down to just their barebones representation.

Usage analysis and its warnings can uncover hidden inefficiencies in the code, such as computing with indices instead of the actual data, and thus it also encourages better structure of code. Erasure also seems to work seamlessly with proof terms built with tactics.

Our erasure approach is easily adaptable to other languages, although we would expect it to be less useful in non-dependently typed languages as functions there generally do not have runtime-irrelevant arguments.

8.1 Limitations

While our erasure algorithm works well in practice, there is a number of limitations which we would like to address in future where possible.

Currently, our scheme cannot erase from arguments of higher-order functions because functional arguments do not have global names that could be used to form nodes. In future work, we plan to use nested nodes (i.e. where the node f_{i_j} stands for the j-th argument of the i-th argument of f) to address this issue. In the meantime, we have developed a work-around for an important special case: type classes (Section 4.4.2).

We also cannot erase from lambdas because, again, they do not have a global name in $\mathsf{IR}_{(\Box)}$. A solution might be a smarter analysis, using the principles of lambda lifting.

Our approach requires whole-program analysis and compiling code separately would be an interesting research problem. Possible approaches include generating all consistent erasure-variants for every function (although this would probably suffer from a combinatorial explosion due to the number of factors involved, especially data constructors) or fully explicit annotations to fix the desired erasure-variant.

Usage analysis requires that the program is well-typed, passes all checks and contains the main function. This means that it is currently impossible to analyse unfinished programs and libraries. A way to enable analysis of libraries would be a suitable set of compiler pragmas conveniently marking some functions as runtime-relevant without being referenced from main if they are expected to end up in generated code at all.

A limitation which may be more difficult to address is that totality is critical for effective erasure. If a function has non-exhaustive matching, the compiler must insert additional branching to report coverage errors at runtime. Unfortunately, this branching must often inspect arguments that were intended to be erased, thus preventing them from being erased. If, on the other hand, a function is non-terminating, and it is used as an (erased) absurd argument of a function in a relevant context, our implementation may generate a terminating but incorrect program, instead of a non-terminating program. This is a known problem, which has been discussed in at least Mishra-Linger's dissertation [16], with a similar problem discussed in Coq's extraction literature [12].

Since neither termination nor falsity is decidable, it is not possible to determine whether the argument being erased is safe to remove without causing the above outcome. Therefore, we must assume that no application of a non-total function can be removed from the program and forbid erasure along the path in the data flow graph all the way from the non-total application to $\mathsf{main}_\star$.

Since usage patterns are inferred, the same function can have different usage pattern in different programs (if the function is in a library, it can be regarded as a limited form of erasure polymorphism). However, within a single program, the usage pattern for any global name is fixed.

This presents problems with frequently used types such as (dependent) pairs, which are often used in different ways in different contexts: since one context uses just the first component and another one uses just the second component of pairs, we cannot erase either. Currently, this requires manually splitting the types. The dependent pair type in Idris has been split into Sigma, Exists and Subset. Instead, we would like the compiler to detect disjoint usages of global names and generate differently erased variants of the same entity.

8.2 Future work

In the near future, we would like to address the limitations discussed in the previous section. For example, we can try to generalise the type class work-around by using recursive nodes in order to erase from arguments of higher-order functions.

Currently, erasure from functions happens by replacing the erased pieces of computation with the undefined value $\Box$. There is an ongoing effort to remove the unused arguments altogether, exactly as we do now with data constructors.

The core language TT was initially designed to be an easily checkable minimal language with dependent types. Now that we have implemented an erasure analysis, we would like to extend this language (and the currently untyped intermediate language $\mathsf{IR}_{(\Box)}$)

with an erasure type system. As well as helping to ensure correctness of erasure, this would proved an *independently checkable* core language incorporating erasure.

Erasure analysis is currently always a whole program analysis. As programs get larger, this may become less practical. We would therefore like to implement a suitable set of compiler pragmas that would allow erasure-checking libraries without needing programs to use them. We do not expect this to pose many difficulties, especially since large parts of independent modules will be private or abstract.

Finally, although we generally advocate writing total functions where possible, it would be very useful to support erasure in partial functions.

8.3 Summary

We have shown that whole-program analysis is a useful way to identify the runtime-irrelevant parts of code while being language-agnostic and effective. We have also shown that usage analysis can be understood in terms of data flow and data dependency graphs, which is a formulation more amenable to reasoning and extensions.

We have shown that programming with views, and dependently typed programming in general, does not have to be inefficient: erasure can recover optimal asymptotic complexities of programs, making it reasonable and practical, even in cases where the methods currently in widespread use fall short. Our method for erasure has been designed and implemented to deal with the kind of programs which arise *in practice* when programming with dependent types, and successfully erases the parts of programs which are intended as static information only.

After all, the most efficient code is code that is never run.

Acknowledgements

We would like to thank Jonathan Leivent, who pointed out many practical concerns in advance (such as usage analysis of separate libraries) and for lots of stimulating and inspiring discussion. We also thank Jan de Muijnck-Hughes for his comments on a draft of this paper.

References

[1] J.-P. Bernardy and G. Moulin. Type-theory in color. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 61–72, New York, NY, USA, 2013. ACM.

[2] A. Bove and V. Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15(4):671–708, 2005.

[3] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013.

[4] E. Brady, C. McBride, and J. McKinna. Inductive families need not store their indices. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs, Torino, 2003*, volume 3085 of *LNCS*, pages 115–129. Springer-Verlag, 2004.

[5] C. Casinghino, V. Sjöberg, and S. Weirich. Combining proofs and programs in a dependently typed language. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 33–45, New York, NY, USA, 2014. ACM.

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

[7] H. Goguen, C. Mcbride, and J. Mckinna. Eliminating dependent pattern matching. In *of Lecture Notes in Computer Science*, pages 521–540. Springer, 2006.

[8] A. Gundry. *Type Inference, Haskell and Dependent Types*. PhD thesis, University of Strathclyde, 2013.

[9] A. Gundry and C. McBride. Phase your erasure. 2013.

[10] J. Leivent. Erasable relevance. `https://github.com/jonleivent/mindless-coding/blob/fd2d662381aa7a805c73ac2bfeb1f1cadcfea47a/erasable_relevance.v`, accessed on 2015-02-28., 2014.

[11] J. Leivent. Mindless coding. `https://github.com/jonleivent/mindless-coding`, accessed on 2015-02-28., 2014.

[12] P. Letouzey. A new extraction for Coq. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs*, volume 2646 of *Lecture Notes in Computer Science*, pages 200–219. Springer Berlin Heidelberg, 2003.

[13] P. Letouzey. Coq extraction, an overview. In C. D. A. Beckmann and B. Löve, editors, *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*, volume 5028 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008.

[14] P. Letouzey and B. Spitters. Implicit and noncomputational arguments using monads, 2005.

[15] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[16] R. N. Mishra-Linger. Irrelevance, polymorphism, and erasure in type theory, 2008.

[17] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[18] C. Paulin-Mohring. Extracting f(omega)'s programs from proofs in the calculus of constructions. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 89–104, 1989.

[19] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. In *Theoretical Computer Science*, pages 203–217. ACM Press, 1999.

[20] The Agda authors. *Agda Wiki: Irrelevance*, 2014. Accessed on 25 Feb 2015.

[21] P. Wadler. Efficient compilation of pattern matching. In *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.