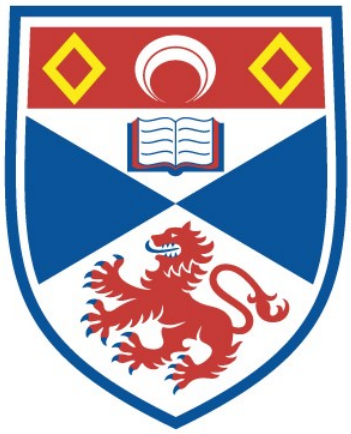
Erasure in dependently typed programming

Matúš Tejiščák

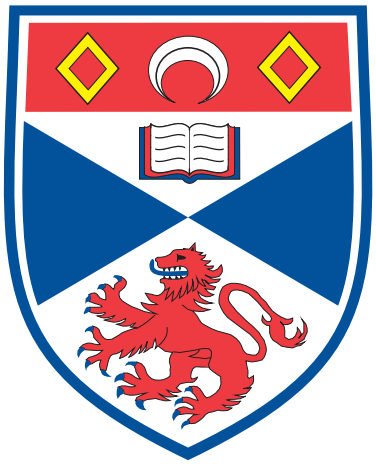
A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



2020

Erasure in Dependently Typed Programming

Matúš Tejiščák



University
of
St Andrews

This thesis is submitted in partial fulfilment for the degree of PhD at the University of St Andrews

March 2019

Candidate's declaration I, Matúš Tejiščák, do hereby certify that this thesis, submitted for the degree of PhD, which is approximately 80,000 words in length, has been written by me, and that it is the record of work carried out by me, or principally by myself in collaboration with others as acknowledged, and that it has not been submitted in any previous application for any degree.

I was admitted as a research student at the University of St Andrews in September 2013.

I received funding from an organisation or institution and have acknowledged the funder(s) in the full text of my thesis.

12 June 2020

Date and Signature of candidate

Supervisor's declaration I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of PhD in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

Date and Signature of supervisor

Permission for publication In submitting this thesis to the University of St Andrews we understand that we are giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. We also understand, unless exempt by an award of an embargo as requested below, that the title and the abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker, that this thesis will be electronically accessible for personal or research use and that the library has the right to migrate this thesis into new electronic forms as required to ensure continued access to the thesis.

I, Matúš Tejiščák, confirm that my thesis does not contain any third-party material that requires copyright clearance.

The following is an agreed request by candidate and supervisor regarding the publication of this thesis:

Printed copy No embargo on print copy.

Electronic copy No embargo on electronic copy.

12 June 2020

Date and Signature of candidate

Date and Signature of supervisor

Underpinning Research Data or Digital Outputs I, Matúš Tejiščák, hereby certify that no requirements to deposit original research data or digital outputs apply to this thesis and that, where appropriate, secondary data used have been referenced in the full text of my thesis.

12 June 2020

Date Signature of candidate

Abstract

It is important to reduce the cost of correctness in programming. Dependent types and related techniques, such as type-driven programming, offer ways to do so.

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 linear-time algorithms run in exponential time, or worse. It would be unnacceptable, and contradict our goal of reducing the cost of correctness, to make programs run slower by only describing them more precisely.

Current systems cannot 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.

This dissertation explains why we need erasure, that it is different from other concepts like irrelevance, and proposes two ways of erasing non-computational data. One is an untyped flow-based useless variable elimination, adapted for dependently typed languages, currently implemented in the Idris 1 compiler.

The other is the main contribution of the dissertation: a dependently typed core calculus with erasure annotations, full dependent pattern matching, and an algorithm that infers erasure annotations from unannotated (or partially annotated) programs.

I show that erasure in well-typed programs is sound in that it commutes with single-step reduction. Assuming the Church-Rosser property of reduction, I show that properties such as Subject Reduction hold, which extends the soundness result to multi-step reduction. I also show that the presented erasure inference is sound and complete with respect to the typing rules; that this approach can be extended with various forms of erasure polymorphism; that it works well with monadic I/O and foreign functions; and that it is effective in that it not only removes the runtime overhead caused by dependent typing in the presented examples, but can also shorten compilation times.

Acknowledgements

To my parents.

Milí rodičia,
ďakujem, že ste mi dali fantastický štart do života s prvotriednym vzdelaním
a všetkým ostatným, aj keď vy ste také možnosti nemali.

I'm very grateful to Dr Edwin Brady, who supervised and advised the thesis. His insights, help, and advice throughout were just indispensable in all aspects of doing the PhD.

I'm also grateful to Dr Andreas Abel, the external examiner, and Dr Christopher Jefferson, the internal examiner and convener, who worked very hard to ensure that I receive the excellent, detailed, and acute feedback that made the dissertation so much better.

I'd like to thank the office mates from JC0.16: Adam D. Barwell, Chris Schwaab, and Franck Slama for the lots of whiteboard discussions, reading groups – and the company in the trenches in general. Same goes for other people "across the corridor", Jan De Muijnck-Hughes, David Castro, Markus Pfeiffer, Özgür Akgün.

Many thanks to Jonathan Leivent for the lots of e-mails we've exchanged about erasure and its ergonomy.

I wouldn't have ended up in this place if Flu had not nudged me onto this trajectory. Thank you, Flu.

I'd like to thank the incredible Staartango people and JazzWorks for providing the necessary work-life balance.

My dearest loveliest Elizabeth, thank you for your love and support along the way.

This work was supported by the University of St Andrews (School of Computer Science).

Contents

List of Theorems

List of Figures

List of Tables

Chapter 1

Introduction

Software failures are a problem. Software failures can be expensive, not only economically but also in terms of wasted scientific potential or even human lives – whether it is the Mars Climate Orbiter that (likely) disintegrated in the Martian atmosphere because of unit mismatch between two systems [ZBM99]; the Therac-25 radiation therapy machine that killed several patients by radiation overdose caused by a race condition [LT93]; or the recently well publicised error where a cryptographic library would leak private keys and other secrets from the process memory in response to malformed network requests [Dur+14].

Software is everywhere. Our society relies on computers more and more every year. Countries are gradually phasing out cash or paper train tickets in favour of electronic systems; people replace their watches with tiny computers that also happen to be able to show time, and forgo their maps in favour of satellite navigation systems, which might eventually be obsoleted by self-driving cars, anyway.

Therefore, problems lurk everywhere. If computers are going to run all aspects of our lives, we must ensure that they do what we intend (and since computers do precisely what we tell them to do, this is actually a problem of ensuring that we say what we mean). In other words, we need reliable software since the impact of software failures, like those mentioned above, will only be amplified by the breadth of deployment of any affected software.

There are ways to deal with it. Of course, people have recognised this problem and created methods and tools for writing software. These include development methodologies, testing, static checkers and analysers, type systems, programming paradigms, proof assistants, formal verification frameworks, and other, that would help achieve the desired level of reliability.

But the cost matters, too. However, correctness is only one aspect of software. Software development happens under various (often contradictory) constraints and rather than striving for correctness, companies usually minimise *cost*, where software failure is just one of many sources of cost, another one being development effort.

Therefore, it is not sufficient to devise a methodology which is 100% reliable but at the same time so expensive to apply or otherwise impractical – such as producing too slow programs – that noone will use it. We generally want to *reduce the cost of writing correct software*, and thus reliability should not be too expensive in other respects.

1.1 Reliability and productivity through static typing

Statically typed programming languages provide guarantees and aid productivity of programmers by giving them:

- Means to *express* some properties of programs in their types ("this variable called `age` will never have the value `"hello"` or `null`").

  This allows programmers to state their intentions.

- Means to *assist* the programmer with writing programs that satisfy the given types, by code completion or more advanced assistance.

  This helps programmers follow their stated intentions and reduces the effort necessary to do so.

- Means to *check* the adherence of the resulting programs to their types *before* executing them and to report an error at compile time if any of the properties expressed by the types are violated.

  This verifies that the resulting program complies with the programmer's stated intentions.

The second point is often overlooked. Contrary to widespread belief, static typing is not only about the machine requiring the programmer to put type annotations in the program and then telling the programmer off when they get the types wrong.

With static types, the computer has a lot of information about the program, even if it is still being written and incomplete, and it can use that information to assist the programmer. Already in relatively simply typed languages like Java, we take features like code completion for granted. Yet stronger type systems have even more potential for programmer assistance:

- Systems like Agda [Nor07; Agd17a] or Idris [Bra13] can infer (some) *programs from types* – if we explain the plan to the computer in high-level terms which are sufficiently precise given the context, we can leave it to fill in the boring details for us, correctly.

  Already in Haskell [Jon03], we can have code generated by giving a type [Gun13]. For example, we can obtain different behaviours from the same source code by choosing, the desired typeclass instance in its type: interpreting a `MonadPlus` program using the list instance and the `Maybe` instance will lead to quite different programs.

- If the program cannot be inferred entirely automatically, perhaps because we decided that a less precise type is sufficient, types can still interactively guide program development; not only by providing code completion, but also by advanced assistance with case analysis, partial proof/program search, program transformation and refactoring, etc.

  This gives rise to methodologies like type-driven development [Bra17] or even "mindless coding" [Lei14b], where the programmer states the *type first*, and then carries out an interactive dialogue with the computer to find a program that satisfies the stated type, possibly ending up refining the type and repeating the process.

- Types are a form of computer-checked and computer-enforced documentation.

- Types can help produce more performant code by proving the absence of certain behaviours.

It is true that Hindley-Milner-style systems can (mostly) infer types from programs, eliminating or reducing the need of specifying the types explicitly. However, as argued above, it is likely more useful to derive the program from the intention rather than the intention from the program.

For that, however, we need a sufficiently strong type system that is able to express the intention.

1.1.1 Reliability and productivity through dependent types

Different static type systems form a whole scale of expressivity and precision. Above, I mentioned that static types let programmers state their intentions. The strength of a type system determines how precisely programmers are allowed to express these intentions.

1.1.1.1 Untyped languages

Statically untyped languages, like Scheme or Perl with `use strict`, will not check the types of variables but will at least check their existence. This could be interpreted as giving the same type to each value [Har16, Sec. 17.4].

Type signatures do not really apply here but we can at least contrive one. The type of a list reversal function says nothing beyond the fact that `reverse` has been defined.

```
reverse : Value
```

1.1.1.2 Simple types

Languages like C or (pre-generics) Java permit giving *simple types* to values, where an example of a simple type is `int`, `String`, or `ArrayList`. Such type systems can already express that we cannot multiply `"hello"` with `"world"`, for example.

The type of reverse gets slightly more interesting.

$$\mathsf{reverse} : \mathsf{List} \longrightarrow \mathsf{List}$$

We can tell that reverse is a function and it takes a list and returns a list. What's the type of the elements of the list, this type signature does not say.

1.1.1.3 Parameterised types

A parameterised type, such as ArrayList⟨String⟩ in Java or Map String Int in Haskell gives us more precision than simple types – we can now express properties like the types of elements stored in containers.

This leads to a more precise type of reverse.

$$\mathsf{reverse} : (a : \mathsf{Type}) \longrightarrow \mathsf{List}\ a \longrightarrow \mathsf{List}\ a$$

This type signature says, besides other things, that reverse preserves the type of elements contained in the list. It does it by taking an extra type argument a, which is then used in two different places to express that the two list element types are actually the same type, a. Current systems, such as Haskell or Idris, erase all type arguments before code generation so this extra argument does not incur overhead at runtime.

1.1.1.4 Dependent types

Languages with dependent types, such as Idris or Agda, go even further and let us write the type of reverse as follows.

$$\mathsf{reverse} : (a : \mathsf{Type}) \longrightarrow (n : \mathbb{N}) \longrightarrow \mathsf{Vect}\ n\ a \longrightarrow \mathsf{Vect}\ n\ a$$

Compared to the previous version, the type signature has been extended with another argument, $(n : \mathbb{N})$. Furthermore, the type of List a, which talked only about the type of its elements, has been replaced with Vect n a, which also talks about n, the *length* of the list. Lists with length-indexed types are traditionally called *vectors*.

Therefore, this type signature is more precise – it says that reverse preserves also the *length* of the list.

The name "dependent" in dependent types comes from the fact that the *type* of the vector argument (and the return value) of reverse depends on the *value* of its second argument.

$$\begin{aligned}
&\mathsf{reverse}\ \mathsf{Int}\ 0\ [] && \text{— } \textit{accepted by typechecker}\\
&\mathsf{reverse}\ \mathsf{Int}\ 3\ [1,2,3] && \text{— } \textit{accepted by typechecker}\\
&\mathsf{reverse}\ \mathsf{Int}\ 2\ [1,2,3] && \text{— } \textit{rejected by typechecker}\\
&\mathsf{reverse}\ \mathsf{Int}\ 42\ [] && \text{— } \textit{rejected by typechecker}\\
&\mathsf{reverse}\ \mathsf{Int}\ n\ xs && \text{— } \textit{depends on the context}
\end{aligned}$$

Expressions like (reverse Int 2 [1, 2, 3]) are type-incorrect because function (reverse Int 2) expects a value of type (Vect 2 Int) for any a but [1, 2, 3] has the incompatible type (Vect 3 Int).

Strictly speaking, Vect n a also depends on the value of the first argument of reverse. However, the first argument of reverse is a type, and the term "dependent" is usually reserved for type systems where types can depend on any kind of value, not just types.

Dependent types can be much more precise than shown in these examples, which means that dependently typed languages can also provide very precise guarantees about programs written in them.

Lightweight verification We need not strive for maximum precision all the time and prove all our programs completely correct. In line with the theme of "reducing the cost of correctness", the optimal amount of precision depends on the tradeoff between the effort expended and the guarantees it buys. Ordinary programs will likely be best served by lightweight verification [SSW10].

Being more declarative Finally, as already mentioned, it is not just the guarantees. The types-first approach of type-driven development, program inference (as opposed to type inference), and programming as an interactive dialogue between the type-checker and the programmer, shift the approach to programming even further from the *how* to the *what* and, with dependent types, also to the *why*.

> With dependent types, we can express our *understanding*, not just our *procedure*. That is the very purpose of declarative programming – to make it more likely that we mean what we say by improving our ability to say what we mean. — Conor McBride [McB03]

1.2 Runtime cost of expressive types

Let's have a look at the declaration of reverse again.

$$\mathsf{reverse} : (a : \mathsf{Type}) \to (n : \mathbb{N}) \to \mathsf{Vect}\ n\ a \to \mathsf{Vect}\ n\ a$$

Why do we have to give the length of the list to reverse in a separate argument, and not just the list itself? Surely the length should be determined by the list. Furthermore, do we need to compute the length of the list before we call reverse, which means extra work of traversing the entire linked list?

Exactly like the type argument a, the length argument n is not necessary for computation because we can implement reverse just by "looking at" the list. We added the argument $(n : \mathbb{N})$ only to introduce a name of a suitable type, which we then used it in two different places: the length index of the last argument and the return value of reverse, to express that these two lengths are in fact the same. The only use of n is

therefore in the type signature and this argument is useless in the implementation of reverse.

However, since n is not a type (it is a natural number), it is not erased trivially by current systems and the argument is present in the compiled code, together with all code that calculates the length of the list before calling reverse.

Thus, already in this simple case, we are getting unnecessary runtime overhead from useless arguments, which cost both memory and time because the callers have to compute and store them.

1.2.1 Linear algorithms taking exponential time

The real problem comes when such typechecking-only information becomes *asymptotically* bigger than the useful information.

In dependently typed languages, natural numbers are usually represented in *unary* – essentially as linked lists with no payload and with length equal to the represented number. The advantage of this representation is convenient inductive reasoning about natural numbers – but they are not very efficient.

We may therefore also want to define *binary* numbers that would be represented as sequences of bits. Since we want to have guarantees about operations on the binary numbers, we index the family of binary numbers with the natural number they represent, so that (Bin n) is the type of binary numbers that represent the natural number n.

```
data Bin : ℕ → Type where
  N : Bin 0
  I : (n : ℕ) → Bin n → Bin (1 + 2*n)
  O : (n : ℕ) → Bin n → Bin (0 + 2*n)
```

With the above definition of binary numbers, the type of a binary adder is expressed as follows.

```
add : (n : ℕ) → (m : ℕ) → Bin n → Bin m → Bin (n + m)
```

The problem with this function is that besides the two binary numbers to add, it also takes the two *unary* forms of the two numbers, called n and m, which are *exponential* in size, compared to the expected space complexity of binary numbers. The code that calls add has to allocate an exponential amount of memory (and spend at least exponential time) to provide the two indices n and m for the function add. Besides that, it will likely have to perform arithmetic on the unary indices to compute the index of the result, too. Finally, exponentially sized unary numbers are also stored in each instance of constructors I and O.

All this makes our program run very slow and use a lot of memory. A program that adds two binary numbers certainly ought to take linear time in the number of

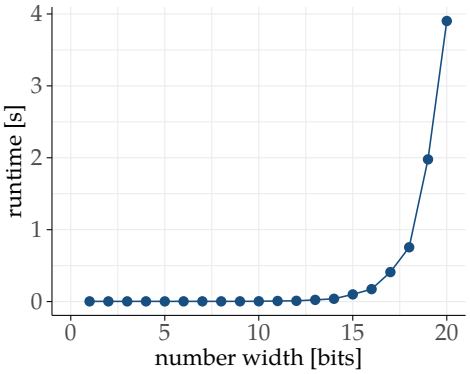


FIGURE 1.1: Runtime of the unerased binary adder (10^5 iterations)

bits. Figure 1.1 illustrates the exponential behaviour of an implementation compiled without erasure.

1.2.2 Summary

The above happens despite the fact that the function add does not care about those unary numbers at all because it works only with their binary representations.

Recall that in the definition of reverse, we added the argument n solely to be able to express that the input length and output length of reverse are the same – but that number is not needed for computation of reverse.

Furthermore, the *implementation* of add does not matter. We extended its *type signature* because we needed the names n and m to represent the denotations of our binary numbers – but this is enough to necessarily change the behaviour of the program from linear to exponential. In other words, the *behaviour* of a program changes depending on what we *say about* it!

This is very bad. We cannot propose dependent types as a way to reduce the cost of writing correct programs if the very mechanism that gives us correctness also introduces serious inefficiencies into the resulting program, such as making an $O(n)$ algorithm take $O(2^n)$ time and space.

Currently used dependently typed languages do not deal with this problem satisfactorily. The Prop universe in Coq is not useful for erasure of indices, such as the index of Bin shown above, neither is the irrelevance of Agda. Zombie [Sjö15] has a form of irrelevance that is able to erase this particular example but equating the distinct concepts of irrelevance and erasure brings its own problems. Further discussion and more examples are given in Sections 3.1.2 and 3.2.

1.3 Thesis

Erasure in practical dependently typed programming is useful and feasible.

Specifically, this means:

Erasure is useful There are dependently typed programs that are elegant and idiomatic but inefficient without erasure.

Furthermore, there are whole programming techniques, such as programming with dependent views, that require erasure to become practical.

Irrelevance is stronger than erasability, which makes it too restrictive to be the only erasure mechanism.

Erasure is feasible There are algorithms that discover and erase non-computational data from such programs.

These algorithms are effective, reasonably efficient, and they are applicable to a real-world implementation of a practical dependently typed programming language.

They are also sound, preserving the meanings of programs.

I return to these claims in Section 9.3 of the concluding chapter.

1.4 Contributions

This dissertation makes the following contributions:

- I give an introduction to dependently typed programming with views and accessibility predicates (Chapter 2).
- I give three different small and self-contained test programs that are idiomatic but inefficient. I explain where the inefficiencies come from, explain that efficiency is not easy to achieve with current approaches and show that erasure can help (Section 3.1.2).

  To my knowledge, this is the first in-depth explanation of why erasure matters and why *proof* erasure is not sufficient.

  These programs are also usable for evaluation of solutions.
- I show that erasure is distinct from irrelevance and useful as a separate concept even in the presence of irrelevance (Section 2.1.7), that erasure is easier to infer than irrelevance, and outline how to extend erasure inference to additionally support irrelevance (Section 7.6).
- I present a non-type-based erasure inference algorithm, a form of flow-based useless variable analysis, that is simple and limited, but also applicable as an external optimisation to a range of dependently typed languages without changing their core calculus (Chapter 4).
  - I show that this erasure approach is effective in removing non-computational data in the examples given in this dissertation, without any explicit erasure annotations (Section 4.6).

  - I describe extensions alleviating some of the limitations of the simple erasure approach, including support for type classes, higher-order functions, and better error reporting (Section 4.5).

  This erasure approach is currently implemented in Idris and has been running on all programs for the past 4.5 years. The prototype compiler[1] of Idris 2, itself implemented in Idris, also relies on erasure. This demonstrates the practicality of this erasure approach.

- I present a dependently typed calculus with optional erasure annotations and full dependent pattern matching via pattern clauses (Chapter 5). Furthermore:

  - I provide an erasure inference algorithm for the calculus. This algorithm does not require any (but permits) user-provided erasure annotations (Chapter 6).
  - I show that the presented erasure inference algorithm is sound and complete with respect to the typing rules (Section 6.6).
  - I describe several different constraint solving algorithms specialised for erasure inference with different tradeoffs of implementation complexity vs. solving performance (Section 6.5.2).
  - I show that unrestricted erasure inference on constructor fields requires whole-program analysis and describe several ways to make analysis more modular (Section 7.4).
  - I prove soundness of the erasure approach in the sense that erasure commutes with reduction in well-typed programs (Section 5.7.10.2).
  - I prove other useful properties of the calculus, such as Subject Reduction (Section 5.7.9), assuming Church-Rosser (Conjecture 5.1).
  - I describe erasure polymorphism of functions (Section 7.3.1) and give a corresponding erasure inference algorithm, and I outline approaches to erasure polymorphism of type families (Section 7.3.2).

    Erasure polymorphism is elaborated into the plain, erasure-monomorphic core calculus, and is thus checkable using the standard typing rules.
  - I show that this erasure approach works well with features like monadic I/O and foreign-function interfaces.
  - I demonstrate the effectiveness of this erasure method on several example programs, showing that besides producing asymptotically faster programs, erasure can also lead to shorter compile times (Section 9.1).

  This calculus is implemented in a small compiler[2], separately from the current version of Idris.

[1] https://github.com/edwinb/Blodwen/
[2] https://github.com/ziman/ttstar/

Chapter 2

Background

This chapter provides a brief introduction to dependently typed programming in Idris, assuming knowledge of typed functional programming ("a user who has written some Haskell or ML").

Section 2.1 introduces dependently typed programming in general, while Section 2.2 introduces dependently typed idioms and programming techniques, from logic and propositional equality, to programming with dependent views.

2.1 Functional programming with dependent types

2.1.1 Syntax

In this chapter, I will use the surface language of Idris [Bra13], which is a dependently typed pure functional language with a Haskell-inspired syntax.

An Idris program is a sequence of data type definitions and function definitions. Each definition is accompanied by a type signature.

```
data ℕ : Type where
  Z : ℕ                                    (2.1)
  S : ℕ → ℕ
```

The above data type definition defines the type of unary natural numbers, $\mathbb{N}$, with two constructors: Z and S. It postulates that $\mathbb{N}$ is a type, that the constructor Z constructs an element of that type ("zero is a natural number"), and that the constructor S maps any element of $\mathbb{N}$ to another element of $\mathbb{N}$ ("a successor of a natural number is a natural number").

Idris supports numeric literals for natural numbers, which are expanded into unary numbers. I will use this syntax sugar as well to write 4 for S (S (S Z))), for example.

The name of the type of types is Type and Idris has a cumulative hierarchy of universes but there is no way to give universe levels explicitly or quantify over them; they are always determined by constraint solving during compilation.

Putting infix operators in parentheses makes them ordinary, prefix names. Therefore (+) m n is equivalent to $m + n$. The following is thus a definition of addition on

natural numbers.

$$\begin{aligned}
&(+) : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
&(+) \quad \mathsf{Z} \quad n \;=\; n \\
&(+) \;\; (\mathsf{S}\; m) \;\; n \;=\; \mathsf{S}\; (m + n)
\end{aligned} \tag{2.2}$$

The (+) notation generalises to *operator sections*, where a partially applied operator can be surrounded by parentheses to yield the corresponding function. For example, (+1) is a function with type $\mathbb{N} \to \mathbb{N}$ that maps a number n to $n + 1$.

Functions are curried by default and they are defined by pattern matching on their arguments. The type signature of the function (+) says that (+) takes two natural numbers and returns a natural number. Its definition consists of two pattern matching *clauses*, where each clause contains patterns on the left hand side and a term on the right hand side.

We can also define *parameterised* types, such as the type of lists that have elements of a given type. Like in Agda or ML, but unlike in Haskell, we write (List a) rather than [a].

$$\begin{aligned}
&\textbf{data}\ \mathsf{List} : \mathsf{Type} \to \mathsf{Type}\ \textbf{where} \\
&\quad \mathsf{Nil} \;:\; \mathsf{List}\ a \\
&\quad (::) \;:\; a \to \mathsf{List}\ a \to \mathsf{List}\ a
\end{aligned} \tag{2.3}$$

The constructor Nil stands for empty lists of any type, and the infix constructor :: prepends an element to a list. Unlike in Haskell, we use ":" for type signatures and "::" to construct lists.

Whenever a value of an argument is expected to be inferrable from the context, we can make the argument *implicit*, which we denote in the type signature by enclosing the argument in braces. For example, the full types of Nil and (::) are as follows:

$$\begin{aligned}
&\mathsf{Nil} \;:\; \{a : \mathsf{Type}\} \to \mathsf{List}\ a \\
&(::) \;:\; \{a : \mathsf{Type}\} \to a \to \mathsf{List}\ a \to \mathsf{List}\ a
\end{aligned} \tag{2.4}$$

When using definitions with implicit arguments, the implicit arguments become invisible – Nil looks like a nullary constant. The implicit arguments are still present implicitly and filled in by the compiler during the elaboration phase using unification [AP11; GM12; Gun13]. We can always give an explicit value to an implicit argument using braces; the following line shows how to construct an empty list of natural numbers.

$$\mathsf{Nil}\ \{\mathsf{a} = \mathbb{N}\}$$

Then if the implicit a is not given explicitly, Nil alone is equivalent to

$$\mathsf{Nil}\ \{\mathsf{a} = \_\},$$

where _ is a special placeholder which stands for "please infer this from the context". This placeholder can also be used freely in terms, most notably in place of explicit arguments for convenience. If its value is not inferrable, Idris will report an error.

Similarly, implicits can be matched explicitly in patterns, as shown in the following function that returns the type of elements of the given list.

```
elemType : List a → Type
elemType {a = a} xs = a
```

The definition in Listing 2.3 does not use the full type signatures given in Listing 2.4 but they are equivalent. Like Haskell, Idris performs *implicit binding of free variables*, which means that free variables in a type signature are understood to be implicitly universally quantified[1]. Such variables are called *free implicits*, as opposed to *bound implicits*, which are bound using braces.

```
sumSquares : List ℕ → List ℕ
sumSquares Nil = Z
sumSquares (x :: xs)
  = let rest = sumSquares xs
      in square x + rest
 where
  square : ℕ → ℕ
  square n = n * n
```

The definition of sumSquares shows what local definitions look like in Idris; the variable *rest* is let-bound in the enclosing expression, and the auxiliary function square is defined in a **where** block to avoid polluting the global namespace.

Postulates Sometimes, we need to postulate an axiom, such as the law of excluded middle, properties of compiler primitives, or a property that holds but we do not want to prove it fully just yet.

In such cases, we can use the form

```
postulate n : τ
```

instead of a function or data type definition to assume a new value with the name n and the type τ.

Mutual recursion Definitions are not mutually recursive by default. Languages like Idris or Agda provide explicit **mutual** blocks for mutually recursive definitions, as shown in Figure 2.1.

[1] Idris requires that these names start with a lowercase letter.

```
mutual
  even : ℕ → Bool
  even    Z   = True
  even  (S n) = odd n

  odd : ℕ → Bool
  odd     Z   = False
  odd   (S n) = even n
```

FIGURE 2.1: Mutual block

Optional laziness In Idris, functions are strict by default. We can make arguments of functions lazy by giving them a Lazy type.

```
if' : Bool → Lazy a → Lazy a → a
```

Lazy is a specially supported type family defined in the standard library in a way that is equivalent to the following.

```
data Lazy : Type → Type where
  Delay : (val : a) → Lazy a

Force : Lazy a → a
Force (Delay x) = x
```

In practice, Delay and Force are inserted by the elaborator and thus a program using laziness need not invoke them explicitly.

```
if' : Bool → Lazy a → Lazy a → a
if'  True  t  f  =  t
if'  False t  f  =  f
```

The only place where laziness can be *seen* is thus the type.

2.1.2 Dependent types

2.1.2.1 Vectors

Some functions, like head : List $a \to a$, or minimum : List $\mathbb{N} \to \mathbb{N}$, cannot return sensible results in some cases, even if their arguments fully comply with the type signature given. In the case of head and minimum, this happens for empty lists. In Haskell, these functions are defined as partial and attempting to evaluate head Nil will result in a *runtime* error, which shows that the type signatures are not precise enough.

We can prevent these runtime errors by including the length of a list in its type and by modifying the types of head and minimum to require arguments of non-zero lengths. Length-indexed lists are traditionally called *vectors* and Listing 2.5 shows the

definition of the corresponding type family.

$$\begin{aligned}&\textbf{data}\ \mathsf{Vect} : \mathbb{N} \to \mathsf{Type} \to \mathsf{Type}\ \textbf{where}\\ &\quad \mathsf{Nil} \ : \ \mathsf{Vect}\ \mathsf{Z}\ a\\ &\quad (::) \ : \ a \to \mathsf{Vect}\ n\ a \to \mathsf{Vect}\ (\mathsf{S}\ n)\ a\end{aligned} \tag{2.5}$$

Compared to the definition of lists, the type constructor Vect has an extra argument – the length – besides the type of elements. Since Idris allows overloading of names (with type-directed disambiguation), we can also use the same names for constructors of both lists and vectors.

We call the length argument an *index* because it may restrict which constructors are usable to construct a value of the corresponding type. For example, (::) cannot be used to construct values of the type Vect Z $\mathbb{N}$ – and rightly so: one cannot make a list empty by prepending elements to it.

We call the type argument a *parameter* because it does not interact with the choice of constructors and it is the same in all recursive occurrences of the type family in its constructors, which means that it applied uniformly throughout the data structure.

The distinction between parameters and indices [Dyb94] is syntactic in some languages (Agda), while others infer it automatically (Idris).

Now we can define an improved head function – the trick is to require a vector with the length of the form (S n), which ensures that the vector will be nonempty.

$$\begin{aligned}&\mathsf{head} : \mathsf{Vect}\ (\mathsf{S}\ n)\ a \to a\\ &\mathsf{head}\ (x :: xs) = x\end{aligned}$$

Idris also notices that the constructor Nil could not possibly construct values with length (S n), and thus it does not require a clause dealing with an empty vector. The function head is now total.

2.1.2.2 Dependent types

The fundamental difference between lists and vectors is that our types and type signatures now contain *terms*. For example, Vect (S (S Z)) $\mathbb{N}$ is the type of vectors of two natural numbers, containing the term (S (S Z)), and Vect (m + square n) a is the type of vectors of length m + square n (with elements of type a). This type contains the term (m + square n).

While simply typed languages, like Haskell, have separate *syntaxes* for terms and types, in dependently typed languages, types are first-class; they can freely contain terms and computation, while functions can take, compute, and return types.

The function append is an example of computation occurring in types: the index of the return type is obtained by evaluating the addition function for the two input lengths. There is nothing magical about (+); it is an ordinary, user-defined function,

as shown in Listing 2.2, and it is not built into the compiler.

$$\mathsf{append} : \mathsf{Vect}\ m\ a \to \mathsf{Vect}\ n\ a \to \mathsf{Vect}\ (m + n)\ a$$

As an example of how functions can compute types, we can define the function varsum with a variable arity. The function VarsumTy takes a number – the desired number of arguments – and returns the type of (curried) functions that take the given number of arguments and return a natural number. We write VarsumTy capitalised to indicate that it computes a type.

$$\begin{aligned}
&\mathsf{VarsumTy} : \mathbb{N} \to \mathsf{Type} \\
&\mathsf{VarsumTy}\ \ \mathsf{Z}\ \ = \mathbb{N} \\
&\mathsf{VarsumTy}\ \ (\mathsf{S}\ n)\ \ = \mathbb{N} \to \mathsf{VarsumTy}\ n
\end{aligned}$$

For example, VarsumTy 3 is the type of functions that take three natural numbers and return a natural number.

$$\mathsf{VarsumTy}\ 3 = \mathbb{N} \to \mathbb{N} \to \mathbb{N} \to \mathbb{N}.$$

We can define varsum for any number of arguments given by *nargs*, by having it return a function of type VarsumTy *nargs*.

$$\begin{aligned}
&\mathsf{varsum} : (\mathit{nargs} : \mathbb{N}) \to \mathsf{VarsumTy}\ \mathit{nargs} \\
&\mathsf{varsum} = \mathsf{vsum}\ \mathsf{Z} \\
&\ \ \textbf{where} \\
&\quad \mathsf{vsum} : \mathbb{N} \to (\mathit{nargs} : \mathbb{N}) \to \mathsf{VarsumTy}\ \mathit{nargs} \\
&\quad \mathsf{vsum}\ \ \mathit{acc}\ \ \ \mathsf{Z}\ \ \ = \mathit{acc} \\
&\quad \mathsf{vsum}\ \ \mathit{acc}\ \ (\mathsf{S}\ n)\ \ = \lambda x : \mathbb{N}.\ \mathsf{vsum}\ (\mathsf{plus}\ x\ \mathit{acc})\ n
\end{aligned} \tag{2.6}$$

Internally, the function varsum defines a function vsum that sums the given numbers using an accumulator.

The right hand sides of the clauses in the definition of vsum have all different types – for $\mathit{nargs} = \mathsf{Z}$, the function returns a natural number, but for $\mathit{nargs} \neq \mathsf{Z}$, it returns a function. In all cases however, the returned value has clearly the type prescribed by VarsumTy.

Named arguments in type signatures In the type signature of varsum, we had to give the first argument of varsum the name *nargs*. We need the name because the return *type* of varsum is calculated from the *value* passed in its first argument – hence *dependent* types.

This approach generalises polymorphic types. In $\mathsf{id} : \{a : \mathsf{Type}\} \to a \to a$, the first (implicit) argument has the type Type, and the *value* of that argument determines the

remaining *type* of id. This is illustrated by partially applying id to its first argument.

$$\mathsf{id}\ \{\mathsf{a} = \mathbb{N}\} : \mathbb{N} \to \mathbb{N}$$

The type of a is Type and values of Type are themselves types – but the mechanism is just the same.

This approach also generalises simple types if the arguments are given names that are unused in the subsequent type signature.

$$(+) : (m : \mathbb{N}) \to (n : \mathbb{N}) \to \mathbb{N}$$

We can use the wildcard symbol to indicate that we are not interested in the names.

$$(+) : (\_ : \mathbb{N}) \to (\_ : \mathbb{N}) \to \mathbb{N}$$

2.1.2.3 Telescopes

Like in other curried-by-default languages, the function arrow symbol associates to the right.

$$a \to b \to c \to d = a \to \big(b \to (c \to d)\big)$$

Since in dependent type signatures, we can bind names, the scope of these names extends to the right of the binder. The name y can therefore be referenced from types c and d, but not from types a or b.

$$(x : a) \to \overbrace{(y : b) \to \underbrace{(z : c) \to d}_{\text{scope of } y}}^{\text{scope of } x}$$

The scopes of x, y, and z are nested like the elements/tubes of a jointed telescope, and we will call any sequence of binders with this property *telescopic*. [Bru91]

2.1.3 Pattern matching

I discuss two major means of pattern matching: pattern clauses and case trees.

2.1.3.1 Pattern clauses

The vector length function is defined using pattern clauses as follows.

$$\begin{aligned}
&\mathsf{vlen'} : (n : \mathbb{N}) \to \mathsf{Vect}\ n\ a \to \mathbb{N} \\
&\mathsf{vlen'}\quad \mathsf{Z}\quad\quad \mathsf{Nil}\quad\ = \mathsf{Z} \\
&\mathsf{vlen'}\ (\mathsf{S}\ k)\ \ (x :: xs)\ = \mathsf{S}\ (\mathsf{vlen'}\ k\ xs)
\end{aligned}$$

This function performs simultaneous matching on two arguments, with the impossible/nonsensical combinations removed. Idris can determine that the two missing clauses are impossible and will accept the definition.

2.1.3.2 Case trees

In its baseline form, a case tree is either a term (when no branching is necessary) or a case split, which names a variable to inspect, called the scrutinee, and contains several (sub-) case trees, called branches. Each branch is labelled with the name of a constructor of the type family of the inspected variable and binds names to be given to the fields projected out of the constructor. Several branches may be omitted in exchange for a single "catch all" branch.

$$
\begin{array}{l}
\mathsf{vlen} : (n : \mathbb{N}) \to \mathsf{Vect}\ n\ a \to \mathbb{N} \\
\mathsf{vlen} = \lambda n.\ \lambda xs. \\
\quad \textbf{case}\ xs\ \textbf{of} \\
\qquad \begin{array}{lcl}
\mathsf{Nil} & \Rightarrow & \mathsf{Z} \\
(::)\ y\ ys & \Rightarrow & \textbf{case}\ n\ \textbf{of} \\
 & & \qquad \mathsf{S}\ k \Rightarrow \mathsf{S}\ (\mathsf{vlen}\ k\ ys)
\end{array}
\end{array}
$$

Some languages, like Coq or Haskell, permit labelling case branches with nested patterns instead of just constructor names. This makes them a hybrid of baseline case trees and pattern matching clauses. By the term "case tree" without further qualification, I will always mean the baseline version, and the term "extended case tree" will always mean the hybrid version.

Case trees are more explicit and operationally focussed than pattern clauses. Pattern clauses can contain nested patterns and do not prescribe the order of matching. Case trees have the order of matching inherent in their structure and do not permit nested patterns. They translate to machine code in a very straightforward way.

This makes case trees a natural intermediary between high-level pattern matching definitions (such as pattern clauses) and machine code. There are established procedures of translating pattern matching to case trees [Aug85; Wad87a; Mar08], and Section 7.2.3 shows how to convert case trees back to pattern clauses.

Idris uses pattern matching clauses in the surface language but its intermediate representations use (baseline) case trees. Other languages, especially those from the ML family, use extended case trees, known as **match** expressions, as the primary means of pattern matching, too (for example Coq, Section 2.1.8.2).

Section 7.2 describes variations of case trees used in various languages and the correspondence between case trees and pattern clauses.

2.1.4 Forced patterns

Pattern matching with dependent types [Coq92] is more subtle than with simple types because by matching on some parts of a pattern may yield information about

other parts of the pattern.

In the above example with vector length, we have been able to define the function vlen′ simply using simultaneous pattern matching and having the compiler observe that some combinations of arguments are impossible. This is however somewhat wasteful – if the vector has the form $x :: xs$, we *know* that n must have the form S k for some k without checking it.

Not checking the constructor tag of n seems like a negligible improvement but it is important for two reasons.

- It conveys the intention of the programmer – it should certainly be possible to write a vector length function by looking only at the vector, disregarding its length index. (Or vice versa.)
- It may cascade and further affect the operational behaviour of the program. If k turns out to be "unused" in some way (which will be made more precise in the rest of the dissertation), then the whole index n is unused and could be removed from the program, saving the time that would be necessary to compute it and the space to store it.

  This is similar to inlining, which Peyton Jones and Marlow propose more for its indirect effects than the transformation itself [JM02].

We will use the name *forced patterns* for patterns that are uniquely determined by other patterns. We will write them as $[T]$, where T is any *term*.

T is a term because the pattern could be forced to an arbitrary value and in such cases, simultaneous matching would be impossible. Consider the following example with a data type that represents a split of a list.

```
(++) : List a → List a → List a    — list concatenation operator
(++)  Nil     ys  =  ys
(++) (x :: xs) ys  =  x :: (xs ++ ys)

data Split : List a → Type where
  MkSplit : (ls : List a) → (rs : List a) → Split (ls ++ rs)
```

In the function that returns the left component of the split, the type of MkSplit *ls rs* is Split (*ls* ++ *rs*), and therefore we know that $xs = ls ++ rs$.

```
left✗ : (xs : List a) → Split xs → List a
left✗ (ls ++ rs) (MkSplit ls rs) = ls    — error: not a pattern: ls ++ rs
```

However, (*ls* ++ *rs*) is not a pattern, and therefore it cannot occur on the LHS of a pattern clause[2]. We cannot hope that the compiler would magically "reverse" the (non-injective!) list append function to recover the values of *ls* and *rs*. Furthermore,

[2]Idris *does* accept this definition as it is written; see Section 2.1.4.5.

both pattern variables would appear *twice* on the LHS, making the patterns non-linear. We therefore mark the problematic pattern as forced to obtain a valid definition of left.

```
left : (xs : List a) → Split xs → List a
left [ls ++ rs] (MkSplit ls rs) = ls
```

Finally, we can also give a better definition for vlen. We will have to use the prefix notation for (::), in order to be able to give a name to the implicit variable k.

```
vlen : (n : ℕ) → Vect n a → ℕ
vlen  [Z]      Nil          = Z
vlen  [S k]  ((::) {n = k} x xs) = S (vlen k xs)
```
(2.7)

The implicit {n = k} says that the variable that was named n in the declaration of (::) should be named k in this pattern match.

Forced patterns do not *bind* ("create") variables – they are always entirely composed of names bound/defined elsewhere. In this case, ++ is a global definition, and ls and rs are bound in the match of MkSplit. There is never any flow of information from forced patterns outwards.

2.1.4.1 Forced constructors

Forced constructors, written ⌈n⌉ where n is a constructor name, are used in cases where the surrounding matches prescribe only *which* constructor must appear in the pattern, but they do not force its arguments.

2.1.4.2 Choices with forced patterns

There are other ways to formulate the function vlen given in Listing 2.7. First, we can extract the variable k from the *first* argument of vlen, and make the corresponding implicit argument of (::) forced. The constructor S is still forced.

```
vlen2 : (n : ℕ) → Vect n a → ℕ
vlen2   [Z]      Nil          = Z
vlen2  (⌈S⌉ k)  ((::) {n = [k]} x xs) = S (vlen2 k xs)
```
(2.8)

Instead of splitting on the vector, we can branch on the length index n, which forces the constructors of the vector, and also the implicit argument n of (::). However, matching on the length index does not force x or xs.

```
vlen3 : (n : ℕ) → Vect n a → ℕ
vlen3   Z       ⌈Nil⌉        = Z
vlen3  (S k)  (⌈::⌉ {n = [k]} x xs) = S (vlen3 k xs)
```
(2.9)

For completeness, we can forgo matching altogether, although this function does not follow the structure of the previous three.

$$\begin{aligned}&\mathsf{vlen4} : (n : \mathbb{N}) \to \mathsf{Vect}\ n\ a \to \mathbb{N}\\&\mathsf{vlen4}\ n\ xs = n\end{aligned} \tag{2.10}$$

The definition of vlen4 has got a different structure from the others, but all three other definitions of vector length: vlen in Listing 2.7, vlen2 in Listing 2.8, and vlen3 in Listing 2.9, are almost equivalent – they differ only in which patterns are forced.

Which definition of vector length should one use? Can the machine choose the right one? I will use further examples to argue that it depends on the circumstances and the intention of the programmer.

2.1.4.3 Choice of forced patterns is up to the programmer

If there are multiple options, the choice of which patterns should be forced is up to the programmer. As mentioned at the beginning of Section 2.1.4, for humans, it serves as documentation of the intended operational behaviour of the function; for the machine, it may influence how the code is compiled: for example how patterns are compiled to case trees and which parts of the program are erasable.

While the compiler can provide assistance by using heuristics or pre-defined rules, only the programmer knows the intention of the code.

2.1.4.4 Interpretation of forced pattern choice

Consider the following program, which calculates the half of a natural number, given a proof that the number is even.

$$\begin{aligned}&\textbf{data}\ \mathsf{Even} : \mathbb{N} \to \mathsf{Type}\ \textbf{where}\\&\quad \mathsf{EZ}\ \ : \mathsf{Even}\ \mathsf{Z}\\&\quad \mathsf{ESS}\ : (n : \mathbb{N}) \to \mathsf{Even}\ n \to \mathsf{Even}\ (\mathsf{S}\ (\mathsf{S}\ n))\end{aligned}$$

$$\begin{aligned}&\mathsf{half} : (n : \mathbb{N}) \to \mathsf{Even}\ n \to \mathbb{N}\\&\mathsf{half}\ \ \ \mathsf{Z}\ \ \ \ \ \ \ \ \ \ [\mathsf{EZ}]\ \ \ \ \ \ = \mathsf{Z}\\&\mathsf{half}\ \ (\mathsf{S}\ (\mathsf{S}\ k))\ \ (\lceil\mathsf{ESS}\rceil\ [k]\ kEven)\ = \mathsf{S}\ (\mathsf{half}\ k\ kEven)\end{aligned} \tag{2.11}$$

(The case where $n = \mathsf{S}\ \mathsf{Z}$ is recognised as impossible by Idris and can be omitted.) There is another way to express the same function.

$$\begin{aligned}&\mathsf{half} : (n : \mathbb{N}) \to \mathsf{Even}\ n \to \mathbb{N}\\&\mathsf{half}\ \ \ [\mathsf{Z}]\ \ \ \ \ \ \ \ \mathsf{EZ}\ \ \ \ \ \ = \mathsf{Z}\\&\mathsf{half}\ \ [\mathsf{S}\ (\mathsf{S}\ k)]\ \ (\mathsf{ESS}\ k\ kEven)\ = \mathsf{S}\ (\mathsf{half}\ k\ kEven)\end{aligned} \tag{2.12}$$

The choice of either implementation can be interpreted from related but distinct perspectives.

Proofs vs. indexed views This perspective is related to programmer's intuition about the program and to documentation of intent – the first bullet point from the beginning of Section 2.1.4.

The definition in Listing 2.11 can be interpreted as a function that takes a number, together with a proof that the number is even, and returns the half of the number. When performing the computation, we "look at" the number n but we do not care about the proof of evenness – it is there only to convince the compiler that the clause for n = S Z can be left out because it can see that no value of Even (S Z) can be constructed.

On the other hand, the definition in Listing 2.12 can be interpreted as a function that takes a *view* of a number, indexed by that number, and uses the view to compute the half of the number. In this case, we compute the result by looking at the *view* and the number n is merely an index that has to be present for formal reasons.

In the general case, the index is not just a formality – it mediates the correspondence between the view and other parts of the type signature, like in the following program that proves that all quadruples of a natural number are even.

```
data Quad : ℕ → Type where
  QZ : Quad Z
  QS : (n : ℕ) → Quad n → Quad (S (S (S (S n))))

quadIsEven : (n : ℕ) → Quad n → Even n
quadIsEven     [Z]              QZ          = EZ
quadIsEven [S (S (S (S k)))] (QS k kQuad) = ESS (ESS (quadIsEven k kQuad))
```

Here, the function looks only at the proof of quadrupleness but the index n is still necessary in the type signature to express that both predicates, Quad and Even, talk about the same number, which is n.

Operational behaviour This perspective is related to the operational behaviour of the program at runtime – the second bullet point from the beginning of Section 2.1.4 – and it focuses on the different ways of translating a declarative program (pattern matching clauses) into a more operational description of computation.

In Listing 2.11, the annotations say that the function should inspect its first argument, n, to decide which clause should be used for reduction. The constructor tag of the proof need not be inspected at all, and the only thing we might need from the proof is the recursive subproof. If the subproof turns out to be unused, then the whole second argument ends up being unused, too.

In Listing 2.12, the annotations indicate that the function should inspect its second argument, a *view* of its first argument. The first argument need not be inspected at

all because we can choose the clause solely on the second argument, and the pattern variable k can be projected out of the view as well.

Erasure This perspective is also related to the operational behaviour of the program but with a different focus. Since forced patterns need not be inspected, any value that ends up being matched only with forced patterns (or not at all) could potentially be erased. Because erasure cascades and because a single usage of an entity blocks its erasure everywhere else, forcing chosen in a consistent/concordant way allows erasing much more than randomly distributed forcing annotations – and that is why choice of forcing matters for erasure.

In Listing 2.11, the inspected argument is the number n and the proof of evenness is not inspected at all because:

- in the base case, it is forced entirely;
- in the recursive step, its constructor tag and first argument are forced. The proof *kEven* is referenced from the RHS but only as the second argument to `half`, which, by induction, is not inspected.

Therefore, in Listing 2.11, the second argument of `half` could be erased.

In Listing 2.12, the inspected argument is the *view* and in both clauses, its index, n, is forced entirely. Therefore, in Listing 2.12, the first argument of `half` could be erased.

2.1.4.5 Forced patterns in dependently typed languages

Forced patterns and forced constructors are present in Epigram [MM04; BMM04] under the name *presupposed terms* and *presupposed constructors*.

Agda has forced patterns, called *inaccessible patterns* or, coloquially, *dot patterns*, which are marked with a dot to obtain a pattern like $.(ls +\!\!+ rs)$. Recent versions allow omitting the dot under certain conditions.

Idris has forced patterns under the name *inaccessible patterns*, but there is no way to mark them in the surface syntax. Idris allows unannotated terms on the LHS of pattern clauses and then decides automatically which subterms are forced. If there are multiple valid choices, the implementation will choose one.

$\mathsf{TT}_\star$, the calculus introduced in Chapter 5, has explicit forced patterns and explicit forced constructors.

2.1.5 Totality of recursive functions

A recursive function is total if it computes a result for every possible (well-typed) input in finite time. The benefit of totality is twofold; first, it makes programs safer, with no possibility of crashing with a message like "`Prelude.head: empty list`" – a notorious runtime error from the Haskell standard library. Second, totality of definitions is a prerequisite for consistency of the corresponding logic (see Section 2.2.2), which is

useful when we want to prove properties of our programs within the programming language.

Totality is usually checked separately from type-correctness by a totality checker, whose three major tasks are coverage checking, termination checking and productivity checking.

2.1.5.1 Coverage checking

The coverage checker checks that all possible combinations of constructors have been covered in a pattern matching definition. There are standard methods to perform this check [SP03; GMM06].

Forced patterns Consistency of forced patterns is related to coverage checking, since it can be checked by converting patterns to case trees. I do not give an algorithm to check forced patterns but I will discuss a few examples.

$$
\begin{aligned}
&f : \mathsf{Bool} \to \mathsf{Bool} \\
&f\ [\mathsf{True}] = \mathsf{True}
\end{aligned}
$$

The above function claims that the constructor True is forced in its first argument. Since forced constructors correspond to single-branch case trees (Section 7.2.3.2), converted to case trees, f looks like the following.

$$
\begin{aligned}
&f : \mathsf{Bool} \to \mathsf{Bool} \\
&f = \lambda x.\ \textbf{case}\ x\ \textbf{of} \\
&\qquad\qquad \mathsf{True} \Rightarrow \mathsf{True}
\end{aligned}
$$

A coverage checker can discover that the other alternative for x, False, is type-correct and there is therefore no reason to force x to True.

Another example would be the following function.

$$
\begin{aligned}
&f : \mathsf{Bool} \to \mathsf{Bool} \\
&f\ \ [\mathsf{True}]\ \ = \ \mathsf{True} \\
&f\ \ [\mathsf{False}]\ \ = \ \mathsf{False}
\end{aligned}
$$

Translated to the case tree form, the above function f looks like the following.

$$
\begin{aligned}
&f : \mathsf{Bool} \to \mathsf{Bool} \\
&f = \lambda x.\ \textbf{case}\ x\ \textbf{of} \\
&\qquad\qquad \mathsf{True}\ \ \Rightarrow\ \mathsf{True} \\
&\qquad\qquad \mathsf{False}\ \Rightarrow\ \mathsf{False}
\end{aligned}
$$

Case tree elaboration again reveals that the forcedness annotations in the pattern clauses were inconsistent since a pattern can be forced only with a single-branch case match.

The following function is even worse: different clauses force different arguments.

$$\begin{aligned} &g : \mathsf{Bool} \to \mathsf{Bool} \to \mathbb{N} \\ &g\ \ \mathsf{True}\ \ [\mathsf{True}] &&= 0 \\ &g\ [\mathsf{True}]\ \ \mathsf{False} &&= 1 \\ &g\ \ \mathsf{False}\ \ [\mathsf{True}] &&= 2 \end{aligned} \tag{2.13}$$

There is no elaboration of this function into a case tree form that would translate back to these pattern clauses. This would therefore be discovered in the case tree elaboration phase, again.

The above examples can be seen as arguments for case trees in the core calculus (Section 7.2.1), where inconsistent forcing annotations would be inexpressible.

2.1.5.2 Termination checking

Termination checking ensures that evaluation of a function will eventually terminate in a finite number of steps for all possible arguments.

A simple approach to termination checking is choosing one of the arguments of the recursive function as the *decreasing* argument, and allowing only recursive calls where the argument in the decreasing position is a subterm of that argument in the parent call.

This admits functions like the following, where the decreasing argument is the only (explicit) argument of the function – the list.

$$\begin{aligned} &\mathsf{length} : \mathsf{List}\ a \to \mathbb{N} \\ &\mathsf{length}\ \ \mathsf{Nil}\ \ &&= \mathsf{Z} \\ &\mathsf{length}\ \ (x :: xs)\ \ &&= \mathsf{S}\ (\mathsf{length}\ xs) \end{aligned}$$

On the RHS of the second clause of length, we make a recursive call on the tail of the list, xs, which is a subterm of $(x :: xs)$, and thus the recursive call is allowed.

With this restriction, it is straightforward to rewrite a recursive function using an eliminator/recursor.

More permissive termination checking However, there are many terminating functions that do not follow the formal scheme described above. An example would be the function merge that merges two sorted lists to produce a sorted list.

$$\begin{aligned} &\mathsf{merge} : \mathsf{List}\ \mathbb{N} \to \mathsf{List}\ \mathbb{N} \to \mathsf{List}\ \mathbb{N} \\ &\mathsf{merge}\ \ \mathsf{Nil}\ \ ys\ \ &&= ys \\ &\mathsf{merge}\ \ xs\ \ \mathsf{Nil}\ \ &&= xs \\ &\mathsf{merge}\ \ (x :: xs)\ \ (y :: ys)\ \ &&= \\ &\quad \textbf{if}\ x \leq y \\ &\qquad \textbf{then}\ x :: \mathsf{merge}\ xs\ (y :: ys) \\ &\qquad \textbf{else}\ \ y :: \mathsf{merge}\ (x :: xs)\ ys \end{aligned}$$

This function recurses on subterms but there is no single argument that decreases in every recursive call. So we would like to allow the above, while still disallowing the following.

```
merge' (x :: xs) (y :: ys) =
  if x ≤ y
    then x :: merge' xs (y :: y :: ys)
    else  y :: merge' (x :: x :: xs) ys
```

Dependently typed languages therefore use more permissive termination checking rules, which relax the strict requirement of a single decreasing argument to a different requirement, which admits more programs, such as the requirement that for each function, there is an ordering of its arguments such that the tuple of its arguments in that order decreases lexicographically [Pie01]; using size-change analysis [LJBA01]; using sized types [Abe04; AVW17]; or other approaches [Abe98].

2.1.5.3 Beyond the termination checker

Since termination is undecidable, for any correct termination checker, there will always be programs that are terminating but rejected. The examples of such programs for common dependently typed languages include the functional Quicksort.

```
filter : (a → Bool) → List a → List a
filter p Nil = Nil                                                    (2.14)
filter p (x :: xs) = if p x then x :: filter p xs else filter p xs
```

```
qsort : List ℕ → List ℕ
qsort     Nil     =  Nil                                              (2.15)
qsort  (x :: xs)  =  qsort (filter (≤ x) xs) ++ x :: qsort (filter (> x) xs)
```

In the second clause of qsort above, the first recursive call is performed on the sublist of elements of xs that are smaller-or-equal than x, namely (filter $(\leq x)$ xs), which is not a subterm of $(x :: xs)$. In fact, since filter is a user-defined function, the "sublist" might even be longer than $x :: xs$, if the programmer defines filter as such.

In order to accept the above definition, the termination checker would have to understand that:

- lists have length (even if no notion of length may be present or defined in the program);
- filter p does not increase the length of a list for any p;
- strictly decreasing list length is a sufficient termination argument.

Since filter and List are ordinary definitions within the language, it would have to be able to perform this reasoning in a general manner.

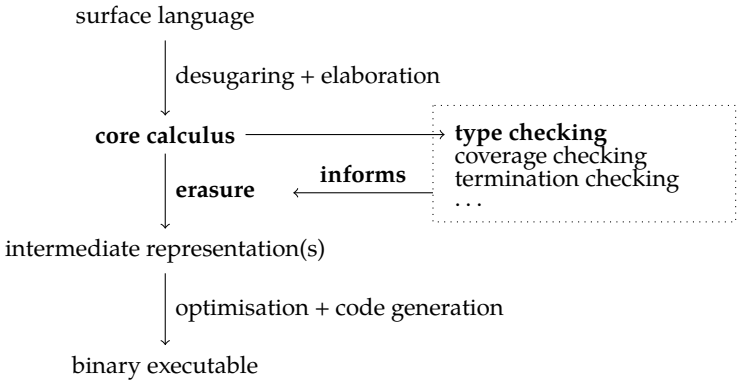


FIGURE 2.2: High-level overview of a typical compilation process, as found in Idris

In general, this situation occurs whenever the "shape" of recursion in the program does not have the same shape as the structure of the input data – the recursion is not *structural*. We therefore need to help the termination checker, either by making the length bounds explicit [Abe04] or in another way.

Several (related) ways to tackle this problem are shown in Sections 2.2.6, 2.2.7, and 2.2.8.

2.1.6 Elaboration

For writing programs, it is very useful to have a rich language with many features that make programmers' lives easier. On the other hand, tasks like type checking may become too complicated to be trustworthy.

One way to deal with the complexity of the surface language is designing a *core language*, which is a simple language into which the rich *surface language* can be translated. This translation is called *elaboration* and for some features, it amounts to just desugaring of syntax; for others, it performs complicated procedures like unification to produce a translation. The corresponding diagram is shown in Figure 2.2.

Idris has a deliberately simple core calculus [Bra13] and its elaboration is responsible for tasks that include the following.

- Making implicit arguments explicit. (The core language of Idris does not have implicits.)
- Filling in implicit arguments in function applications.
- Filling in appropriate patterns for implicit arguments.
- Filling in any other omitted terms (underscore symbols: _)
- Translating interfaces, implementations, and constraint arguments into explicit dictionary passing. (Interfaces and implementations name Idris's flavour of type classes and instances.)
- Filling in terms by proof search.

- Filling in names and types of pattern variables in pattern clauses.
- Filling in type annotations wherever omitted (such as in lambdas).
- Lifting with clauses, case expressions, and related structures (Section 2.2.5) into separate functions.
- . . . and more.

On the one hand, a fully elaborated program is very verbose: all high-level constructs have been implemented using lower-level facilities of the core language, and furthermore the program contains all minute details about the types of all variables and other information necessary for checking and compiling the program.

On the other hand, checking programs expressed in the core language is simpler and therefore more trustworthy than it would be in the surface language.

The surface language can have many features, as long as elaboration explains how to translate them into the core language. Furthermore, elaboration need not be trusted for the program to be type-safe, as long as its result is (reliably) checkable.

In other words, the core language explicitly contains all the evidence needed to efficiently verify that the program is valid.

Elaboration in this chapter Like with the surface language introduced in Section 2.1.1, I will not introduce a formal core language at this point because the purpose of this chapter is to convey intuition rather than to be formal.

For our purposes, it will be sufficient to define "elaboration" as spelling out all implicit arguments in type signatures, applications, and patterns.

2.1.7 Parametricity vs. erasability vs. irrelevance

The notions of parametricity [Rey83; Wad89; BJP12], erasability and irrelevance [Pfe01] are all distinct, even though literature does not always distinguish between them. I will use the terminology as follows.

- With parametricity, functions map related inputs to related outputs.
- With erasability, some values do not affect reduction at runtime.
- With irrelevance, some values do not affect equality.

Abel [Abe11] words the distinction between erasability and irrelevance as *external* erasure (before execution) vs. *internal* erasure (before typechecking).

Parametricity is weaker than erasability Nuyts, Vezzosi and DeVriese [NVD17] present a type theory, ParamDTT, that accepts the definition of an identity function on the naturals $f : \sharp\mathbb{N} \to \mathbb{N}$. This function is parametric but its argument is certainly not erasable. (Their work is further discussed in Section 8.4.2).

Erasability is weaker than irrelevance Consider the following program.

```
data Bool : Type where
  True  : Bool
  False : Bool

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
```

In this program, the argument b of constructor C is erasable because it is not used anywhere 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.

Section 7.6 also argues that irrelevance inference is more complicated than erasure inference alone because of the interaction between irrelevance and typechecking.

Terminology In this dissertation, I make the distinction among:

irrelevant values that can be erased before typechecking;

erasable values that can be erased before runtime;

runtime values that must be preserved until runtime.

2.1.7.1 Irrelevance in terms and types

In a type constructor like Vect : $(n : \mathbb{N}) \to (a : \mathsf{Type}) \to \mathsf{Type}$, both the index n and parameter a should be *relevant*. This is because we want the notion of equality to care about them – the parameter a could have different values, and all of them would lead to different types of vectors. In other words, if we made a irrelevant, the type (Vect (S Z) $\mathbb{N}$) would be definitionally equal to the type (Vect (S Z) Bool). The same applies to the index n and it is clearly undesirable.

However, in a data constructor like the following,

$$(::) : (n : \mathbb{N}) \to (a : \mathsf{Type}) \to (x : a) \to (xs : \mathsf{Vect}\ n\ a) \to \mathsf{Vect}\ (\mathsf{S}\ n)\ a$$

the fields n and a can be made irrelevant. For example, in the case of n, if the tails of two vectors compare equal, it is clear that the length indices will also be equal. Thus it makes sense to avoid comparing them by making them irrelevant.

However, despite the fact that these values are determined by other values, they may not be reconstructible quickly (it is non-trivial to construct the index n from the

tail of a vector) and one may still prefer to *not* make them *erased* at runtime or erase elsewhere instead, as discussed in Section 2.1.4.2.

2.1.7.2 Irrelevance and typed equality

In calculi with typed equality, irrelevance is even more restrictive. Consider the vector length function.

$$
\begin{aligned}
&\mathsf{vlen} : .(n : \mathbb{N}) \to \mathsf{Vect}\ n\ a \to \mathbb{N} \\
&\mathsf{vlen}\ \ [\mathsf{Z}]\ \ \ \mathsf{Nil}\ \ \ \ = \mathsf{Z} \\
&\mathsf{vlen}\ \ [\mathsf{S}\ n]\ \ ((::)\ n\ a\ x\ xs)\ = \mathsf{S}\ (\mathsf{vlen}\ n\ xs)
\end{aligned}
$$

Since the first (explicit) argument of vlen is fully determined by its second argument, it would make sense to mark the argument n of vlen as irrelevant.

Indeed, in ICC* [BB08], the type $\forall(n : \mathbb{N}) \to \mathsf{Vect}\ n\ a \to \mathbb{N}$ would be well formed, thanks to the rule I-Prod, which "resurrects" [Pfe01] the bound variable as relevant on the RHS of a Pi (also known as *context reset* [ML08]).

However, Agda does not accept the type signature $.(n : \mathbb{N}) \to \mathsf{Vect}\ n\ a \to \mathbb{N}$ with a "dotted" (irrelevant) binding of n because n is used as a relevant argument of Vect later in the type. This usage of irrelevant patterns is not consistent with large elimination together with Agda's typed equality and η-equality for records.

In Agda, like in Pfenning's Modal Type Theory [Pfe01], and unlike in the ICC-style calculi [Miq01; BB08; ML08], equality is *typed*: whenever deciding equality, the type checker knows (needs to know) the type that both sides have.

Abel also gives an explicit example of a problematic program [AS12]. I will present the program in the Agda syntax, using the dotted binder $.(b : \mathsf{Bool})$ to denote irrelevance.

$$
\begin{aligned}
&T : \mathsf{Bool} \to \mathsf{Type} \\
&T\ \ \mathsf{True}\ \ = \mathsf{Bool} \to \mathsf{Bool} \\
&T\ \ \mathsf{False}\ = \mathsf{Bool}
\end{aligned}
$$

$$
\begin{aligned}
f\ \ :\ \ &(F : .(b : \mathsf{Bool}) \to (T\ b \to T\ b) \to \mathsf{Type}) \\
&\to (g : F\ \mathsf{False}\ (\lambda z : \mathsf{Bool}.\ z) \to \mathsf{Bool}) \\
&\to (x : F\ \mathsf{True}\ (\lambda h : \mathsf{Bool} \to \mathsf{Bool}.\ \lambda y : \mathsf{Bool}.\ h\ y)) \\
&\to \mathsf{Bool} \\
f\ F\ g\ x &= g\ x
\end{aligned}
$$

Irrelevance makes x a valid argument to g because the erased forms of the type of x and the domain of g, denoted by erasure brackets $\langle\cdot\rangle$, are $\beta\eta$-equivalent:

$$
\begin{aligned}
&\langle F\ \mathsf{False}\ (\lambda z : \mathsf{Bool}.\ z)\rangle = F\ (\lambda z.\ z) =_\eta F\ (\lambda h.\ \lambda y.\ h\ y) \\
&\quad = \langle F\ \mathsf{True}\ (\lambda h : \mathsf{Bool} \to \mathsf{Bool}.\ \lambda y : \mathsf{Bool}.\ h\ y)\rangle
\end{aligned}
$$

The problem is that it is unclear whether we should allow conflation of identity functions at different types and how a typed-equality checker could possibly give a positive answer with different types on both sides [AS12].

Abel also points out that it gets worse if we assume η-equality for the unit type, which makes any two elements of the unit type definitionally equal. In the following program, we change the definition of T True to Unit, and perform a change in the type of x.

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Unit} : \mathsf{Type}\ \textbf{where} \\
\quad \mathsf{MkUnit} : \mathsf{Unit} \\
\\
T : \mathsf{Bool} \to \mathsf{Type} \\
T\ \ \mathsf{True}\ \ =\ \ \mathsf{Unit} \\
T\ \ \mathsf{False}\ \ =\ \ \mathsf{Bool} \\
\\
f\ \ :\ \ \big(F : .(b : \mathsf{Bool}) \to (T\ b \to T\ b) \to \mathsf{Type}\big) \\
\qquad \to \big(g : F\ \mathsf{False}\ (\lambda z : \mathsf{Bool}.\ z) \to \mathsf{Bool}\big) \\
\qquad \to \big(x : F\ \mathsf{True}\ (\lambda h : \mathsf{Unit}.\ \mathsf{MkUnit})\big) \\
\qquad \to \mathsf{Bool} \\
f\ F\ g\ x = g\ x
\end{array}
$$

We can verify that both sides are still $\beta\eta$-erasure-equivalent.

$$\langle F\ \mathsf{False}\ (\lambda z : \mathsf{Bool}.\ z)\rangle = F(\lambda z.\ z) = \langle F(\lambda z : \mathsf{Unit}.\ z)\rangle =_\eta \langle F(\lambda z : \mathsf{Unit}.\ \mathsf{MkUnit})\rangle$$

If this program were to typecheck, the type checker would have to conclude that $(\lambda z : \mathsf{Bool}.\ z)$ is equal to $(\lambda h : \mathsf{Unit}.\ \mathsf{MkUnit})$, which, even ignoring the different types, amounts to concluding that $(z : \mathsf{Bool})$ equals $(\mathsf{MkUnit} : \mathsf{Unit})$. This would mean, by transitivity of equality, that any two Booleans are equal [AS12]. Besides, we get $(\mathsf{const}\ \mathsf{MkUnit} : \mathsf{Unit} \to \mathsf{Unit})$ equal to $(\mathsf{id} : \mathsf{Bool} \to \mathsf{Bool})$ and at this point even the operational behaviour does not match.

We can observe that making the argument only *erased* but not *irrelevant* resolves the problem by rejecting the problematic programs as type-incorrect – from the point of view of type checking, erased b is no different from a runtime argument and it is fully considered in equality.

2.1.8 Implementation

This section briefly discusses some aspects of *implementation* of dependently typed languages.

2.1.8.1 Constructor tags

A data constructor declaration has the following general form.

$$C : (n_1 : \tau_1) \to \ldots \to (n_k : \tau_k) \to T(n_1, \ldots, n_k)$$

where C is the name of the constructor and n_i and τ_i are the names and types of its fields. $T(n_1, \ldots, n_k)$ is the target type, which is always a saturated application of the type constructor of the corresponding type family and may depend on the values n_1 through n_k.

Since the target type, $T(n_1, \ldots, n_k)$, must always have the specific form described above, the target type can never be a functional type, which means that we can define a fixed arity for any constructor. In the above example, the arity of C is k.

If we disallow pattern matching on values with functional types, we can match only on fully applied (saturated) constructors. Saturated constructor applications, with the general form $C\ x_1\ \ldots\ x_k$, can then be represented as $k+1$-tuples of the form $(C, x_1, \ldots, x_k)$, where the first component of the tuple is the name of the constructor.

Constructor tag optimisations If a type family has only one constructor, there is no need to store or check its tag. For single-field constructors, this is known as the *newtype optimisation* [Jon03], which is applicable freely in strictly evaluated languages.

Furthermore, if for a type family, the constructor tag is always determined by the indices of the family, tags can be removed from all constructors of such a family. This is known as the *detagging optimisation* [Bra05] (Section 3.2.1.1).

2.1.8.2 Type checking pattern clauses

The major advantage of pattern clauses for pattern matching is the simplicity of type checking because no complicated (and incomplete) procedures, like unification, are necessary to type check a pattern matching function.

The type checking rule for pattern matching clauses is given by Brady [Bra13]. A fully elaborated function definition has the following general form.

$$
\begin{aligned}
&f : \tau \\
&\Pi^1.\ \underbrace{f\ \overline{P^1}}_{L^1} = R^1 \\
&\quad\vdots \\
&\Pi^n.\ \underbrace{f\ \overline{P^n}}_{L^n} = R^n
\end{aligned}
$$

The definition starts with a type signature for the function f, followed by n pattern clauses. In each pattern clause, Π is an environment that declares the types of pattern variables, L is the LHS of the pattern clause, P_i are the individual patterns for arguments of f, and R is the RHS of the pattern clause.

Given a typing environment Γ, we can check whether a pattern clause is type-correct using the rule Clause.

$$
\frac{\Gamma \vdash \Pi\ \textbf{valid} \qquad \Gamma, \Pi, f : \tau \vdash L : \lambda \qquad \Gamma, \Pi, f : \tau \vdash R : \rho \qquad \Gamma, \Pi \vdash \lambda \approx \rho}{\Pi.\ L = R \quad \textbf{is type-correct}}\ \textsc{Clause}
$$

We check both LHS and RHS in the environment Γ extended with the declarations of pattern variables Π, and possibly by the asserted declaration of f, and then check that the resulting types are convertible. Note that the LHS is checked as a *term*, despite being composed of patterns.

Depending on the programming language, there will be further requirements on valid pattern clauses, especially on the allowed forms of patterns usable on the LHS; for example that the head of every application must be a constructor, that forced patterns must be annotated consistently, and other.

Example: vector length As an example, let us look closer at why the vector length function in Listing 2.7 passes the type check. First, the type signature gives the asserted type of vlen.

$$\mathsf{vlen} : (n : \mathbb{N}) \to \mathsf{Vect}\ n\ a \to \mathbb{N}$$

For brevity, let us write τ for the type of vlen. The first clause of vlen elaborates to the following form.

$$\emptyset.\ \mathsf{vlen}\ \mathsf{Z}\ \mathsf{Nil} = \mathsf{Z}$$

Here, $\Pi = \emptyset$, $L = \mathsf{vlen}\ \mathsf{Z}\ \mathsf{Nil}$, and $R = \mathsf{Z}$. The empty environment Π is trivially valid, and we can see that both L and R have the type $\mathbb{N}$. (Assuming a reasonable environment Γ that defines $\mathbb{N}$, Vect, etc. appropriately.)

$$\frac{\Gamma \vdash \emptyset\ \textbf{valid} \qquad \Gamma, \mathsf{vlen} : \tau \vdash \mathsf{vlen}\ \mathsf{Z}\ \mathsf{Nil} : \mathbb{N} \qquad \Gamma, \mathsf{vlen} : \tau \vdash \mathsf{Z} : \mathbb{N} \qquad \Gamma \vdash \mathbb{N} \approx \mathbb{N}}{\emptyset.\ \mathsf{vlen}\ [\mathsf{Z}]\ \mathsf{Nil} = \mathsf{Z} \quad \textbf{is type-correct}}\ \textsc{Clause}$$

The forced pattern [Z] becomes Z when we check the LHS as a term. The second clause of vlen elaborates to the following.

$$\underbrace{k : \mathbb{N},\ x : a,\ xs : \mathsf{Vect}\ k\ a}_{\Pi}.\quad \mathsf{vlen}\ [\mathsf{S}\ k]\ \big((::)\{\mathsf{n} = k\}\ x\ xs\big) = \mathsf{S}\ (\mathsf{vlen}\ k\ xs)$$

Then the LHS as term, $\mathsf{vlen}\ (\mathsf{S}\ k)\ \big((::)\{\mathsf{n} = k\}\ x\ xs\big)$, is type-correct with type $\mathbb{N}$. The RHS, $\mathsf{S}\ (\mathsf{vlen}\ k\ xs)$, has type $\mathbb{N}$ as well.

$$\frac{\begin{array}{c}\Gamma \vdash k : \mathbb{N},\ x : a,\ xs : \mathsf{Vect}\ k\ a\ \textbf{valid} \\ \Gamma,\ k : \mathbb{N},\ x : a,\ xs : \mathsf{Vect}\ k\ a,\ \mathsf{vlen} : \tau \vdash \mathsf{vlen}\ (\mathsf{S}\ k)\ \big((::)\{\mathsf{n} = k\}\ x\ xs\big) : \mathbb{N} \\ \Gamma,\ k : \mathbb{N},\ x : a,\ xs : \mathsf{Vect}\ k\ a,\ \mathsf{vlen} : \tau \vdash \mathsf{S}\ (\mathsf{vlen}\ k\ xs) : \mathbb{N} \\ \Gamma,\ k : \mathbb{N},\ x : a,\ xs : \mathsf{Vect}\ k\ a \vdash \mathbb{N} \approx \mathbb{N}\end{array}}{k : \mathbb{N},\ x : a,\ xs : \mathsf{Vect}\ k\ a.\ \mathsf{vlen}\ [\mathsf{S}\ k]\ \big((::)\{\mathsf{n} = k\}\ x\ xs\big) = \mathsf{S}\ (\mathsf{vlen}\ k\ xs) \quad \textbf{is type-correct}}\ \textsc{Clause}$$

Forced patterns must be acknowledged The above also explains why this rule will reject an attempt to define vlen as follows.

$$
\begin{aligned}
&\mathsf{vlen} : (n : \mathbb{N}) \to \mathsf{Vect}\ n\ a \to \mathbb{N} \\
&\mathsf{vlen}\ \ n\ \ \ \ \mathsf{Nil}\ \ \ \ = \mathsf{Z} \\
&\mathsf{vlen}\ \ n\ \ (x :: xs)\ = \mathsf{S}\ (\mathsf{vlen}\ \_\ xs)
\end{aligned}
$$

The first clause elaborates as follows.

$$n : \mathbb{N}.\quad \mathsf{vlen}\ n\ \mathsf{Nil} = \mathsf{Z}$$

However, there is no type λ where

$$\Gamma, n : \mathbb{N} \vdash \mathsf{vlen}\ n\ \mathsf{Nil} : \lambda,$$

because the type of $\mathsf{vlen}\ n$ is $\mathsf{Vect}\ n\ a \to \mathbb{N}$ for some abstract n, while the type of Nil is $\mathsf{Vect}\ \mathsf{Z}\ a$, and thus their application is not well typed because of the mismatch between n and Z.

The second clause needs to perform the recursive call with the predecessor of n, which can be obtained either by pattern matching on n, or from the index of (::).

One could argue that if we employed unification, we could find a solution $n = \mathsf{Z}$ automatically – and this is indeed what recent versions of Agda do. However, we want type*checking* as simple and reliable as possible. Having unification and proof search in type checking means a lot more complicated code to trust.

This does not mean that we cannot have unification and the programmer has to write all this information in the program explicitly. The solution $n = \mathsf{Z}$ could be found by a separate elaboration pass *before* typechecking. Then the result of the automated elaboration pass can be incorporated in the program explicitly and checked by an independent, simple and reliable typechecker.

Example: variadic sum Let us have a look at the function vsum from Listing 2.6 and let us do it a little bit less formally than in the previous example. As a reminder, these are the relevant definitions.

$$
\begin{aligned}
&\mathsf{VsumTy} : \mathbb{N} \to \mathsf{Type} \\
&\mathsf{VsumTy}\ \ \ \mathsf{Z}\ \ \ \ = \mathbb{N} \\
&\mathsf{VsumTy}\ \ (\mathsf{S}\ n)\ = \mathbb{N} \to \mathsf{VsumTy}\ n
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{vsum} : \mathbb{N} \to (\mathit{nargs} : \mathbb{N}) \to \mathsf{VsumTy}\ \mathit{nargs} \\
&\mathsf{vsum}\ \ \mathit{acc}\ \ \ \mathsf{Z}\ \ \ \ = \mathit{acc} \\
&\mathsf{vsum}\ \ \mathit{acc}\ \ (\mathsf{S}\ n)\ = \lambda x : \mathbb{N}.\ \mathsf{vsum}\ (\mathsf{plus}\ x\ \mathit{acc})\ n
\end{aligned}
$$

In the first clause of vsum, we typecheck the LHS, which is $(\mathsf{vsum}\ \mathit{acc}\ \mathsf{Z})$, in an environment with $\mathit{acc} : \mathbb{N}$. The variable *nargs* in the type of vsum, corresponding to its second argument, gets the value Z and thus the type of $(\mathsf{vsum}\ \mathit{acc}\ \mathsf{Z})$ is VsumTy Z,

which is definitionally equal to $\mathbb{N}$. The type of the RHS is the type of *acc*, which is $\mathbb{N}$, and thus the types match.

In the second clause of vsum, we typecheck the LHS, which is vsum *acc* (S n), in an environment with $acc : \mathbb{N}, n : \mathbb{N}$. The type of the LHS is thus VsumTy (S n), which is definitionally equal to $\mathbb{N} \to$ VsumTy n.

On the RHS, since the type of vsum (plus x *acc*) n is VsumTy n, the type of the whole lambda expression on the RHS is $\mathbb{N} \to$ VsumTy n, which matches the type of the LHS.

In the previous example, the types of the LHS and RHS were always $\mathbb{N}$. In this example however, the types of the RHS (or LHS) are different in each clause.

Necessity of type signatures for pattern matching functions This method of typechecking pattern matching function definitions relies on knowing the purported type signature for the function being defined.

This usually does not cause problems because in dependently typed languages, top-level definitions are generally required to have type signatures, for several reasons.

- type inference with dependent types is harder than with non-dependent types found in languages like Haskell;
- having top-level type signatures is considered good practice because it is good documentation;
- it enables type-driven development [Bra17] (Section 1.1).

Again, the fact that type*checking* requires type signatures on pattern matching functions does not mean that the programmer has to write all of them – a separate elaboration pass before the typechecking phase is free to infer anything that is necessary.

Correspondence to Coq's extended match An explicit type signature for a pattern matching function can be seen as analogous to Coq's extended pattern matching.

$$
\begin{array}{l}
\textbf{match } T \textbf{ as } n \textbf{ in } \tau \textbf{ return } P(n) \textbf{ with} \\
\mid L_1 \Rightarrow R_1 \\
\quad \vdots \\
\mid L_n \Rightarrow R_n \\
\textbf{end}
\end{array}
\tag{2.16}
$$

Namely, any such **match** expression, as shown in Listing 2.16, can be expressed as an application of a locally bound pattern matching function, shown in Listing 2.17.

$$\begin{aligned}
&\textbf{let } f : (n : \tau) \to P(n) \\
&\quad f\ L_1 = R_1 \\
&\quad \vdots \\
&\quad f\ L_n = R_n \\
&\textbf{in } f\ T
\end{aligned} \tag{2.17}$$

Here, T is the scrutinee of the match expression, and $(n : \tau)$ is a binding of the name that stands for the (possibly complex) scrutinee in the return type $P(n)$, which depends on n. We represent the scrutinee using a single variable in order to be able to substitute for it.

More details on possible uses of local pattern matching definitions can be found in Section 2.2.5.

2.2 Some patterns and idioms of dependently typed programming

2.2.1 Utilities

2.2.1.1 Giving explicit types to terms

It is useful to define a function the, which is an identity function with an explicit type argument.

$$\begin{aligned}
&\mathsf{the} : (a : \mathsf{Type}) \to a \to a \\
&\mathsf{the}\ a\ x = x
\end{aligned} \tag{2.18}$$

The purpose of the is to give explicit type annotations to expressions. This is analogous to expression-level type annotation using (::) in Haskell.

```
the ℕ (1 + 2)      — typechecks with type ℕ
the Type 42        — does not typecheck
the (ℕ → ℕ) id     — typechecks with type ℕ → ℕ
```

2.2.1.2 Finite sets

We can define the type family Fin : $\mathbb{N} \to$ Type, where Fin n stands for a type with n inhabitants.

```
data Fin : ℕ → Type where
  FZ  : Fin (S n)
  FS  : Fin n → Fin (S n)
```

In functional programming	In logic
type	proposition
value of type τ	proof of proposition τ
checking that value P has type τ	checking that proof P proves proposition τ
function arrow	implication
product type	conjunction
sum type	disjunction
Unit	true
Void	false
$\tau \to$ Void	negation of τ
n-ary type constructor	n-ary predicate
nullary type constructor	logical constant
Π type	universal quantification
Σ type	existential quantification
application	modus ponens

TABLE 2.3: Selected parts of the Curry-Howard correspondence

There are no constructors that could construct a value of the type Fin Z: for every number of the form (S n), the type Fin (S n) contains all inhabitants of Fin n prefixed with FS, plus one new inhabitant constructed with FZ.

If we prefer to view Fin n as "the naturals strictly smaller than n", the following embedding may be more fitting.

```
embed : Fin n → Fin (S n)
embed  FZ     = FZ
embed  (FS x) = FS (embed x)
```

The function embed has the same type as FS but it maps the elements differently.

2.2.2 Logic

The famous Curry-Howard correspondence [How80; Wad00; Wad15] states that types of functional programs can be interpreted as propositions and the programs themselves can be interpreted as proofs of these propositions. A brief overview is shown in Table 2.3.

2.2.2.1 Predicates and relations

In Section 2.1.4.4, we saw the type family Even.

```
data Even : ℕ → Type where
  EZ  : Even Z
  ESS : (n : ℕ) → Even n → Even (S (S n))
```

We can observe that the type Even Z is *inhabited*, which means that there exists a value of type Even Z, called EZ. However, there is no way to construct a value of

type Even (S Z). This is because the constructors of Even are designed to make Even n inhabited exactly when n is even.

We can therefore view the type constructor Even as the (unary) *predicate* that a certain number is even, the type Even n as the proposition "n is even", and a value of the type Even n as a proof that n is even.

This allows us to write functions like halve : $(n : \mathbb{N}) \to \mathsf{Even}\ n \to \mathbb{N}$ to restrict usage of halve only to even values of n. With an odd n, we would not be able to invoke the function halve, since we would not have anything to pass in its second argument.

Unit and Void There are two important nullary predicates, which we usually use to model truth and falsity.

```
data Unit : Type where
  () : Unit
```

The type Unit is inhabited by one value, (). In Haskell and ML-style languages, both the type and its inhabitant are called (). In other languages, like Agda or Coq, the inhabitant is traditionally named tt.

There is also the empty type, which represents falsity.

```
data Void : Type where
  — (no constructors)
```

The type Void comes with an eliminating function.

```
voidElim : Void → a
— (no clauses)
```

This function is covering and total because it has a clause for every possible value of its argument[3]. It embodies the "ex falso quodlibet" principle.

Negation is then represented as a function into Void.

```
Not : Type → Type
Not a = a → Void
```
(2.19)

Relations Although the terminology varies, non-unary predicates are often called *relations*. These are encoded as data types in exactly the same way as other predicates.

[3]There are none.

2.2.2.2 Decidable predicates

For decidable predicates, it is useful to define the following type family, which encodes the outcome of a decision procedure.

```
data Dec : Type → Type where
  Yes : (proof : a) → Dec a
  No  : (contra : Not a) → Dec a
```

Unlike Bool, a value of Dec a also contains the proof of the outcome – either a proof of a if the answer is Yes, or a proof of Not a if the answer is No. (Recall from Listing 2.19 that Not $a = a \to$ Void.) In other words, the outcome also contains its *explanation*.

Dec therefore does not suffer from Boolean Blindness [Har11]: if the outcome of the decision procedure were Bool, functions like semiCong in Listing 2.26 would be impossible to implement.

For a decidable predicate $p : a \to$ Type, we can write a decision function that takes a value x and returns the outcome of the check whether predicate p holds for value x.

$$\mathsf{dec}_p : (x : a) \to \mathsf{Dec}\ (p\ x)$$

This will be useful, among other things, with equality (Section 2.2.3.5): instead of having an equality check return just an uninformative Boolean, we can have it return a proof of equality or unequality instead.

2.2.2.3 Consistency

If the existence of a value of type has to guarantee that the corresponding proposition is true, languages have to impose certain restrictions on valid programs.

Like in logic, where a theory is inconsistent if we can prove p and $\neg p$ for a formula p, a type system is inconsistent if it admits a value of type a and $a \to$ Void at the same time. Since having both $(x : a)$ and $(f : a \to \mathsf{Void})$ means that $f\ x :$ Void, the above is equivalent to Void being inhabited, which is the usual definition of inconsistency of a type system.

Totality of functions Consider the following "proof" that 5 is even.

```
even5 : Even 5
even5 = even5
```

The function is type-correct and would be accepted in a language like Haskell (or rather its hypothetical variant with dependent types). However, it does not terminate, and therefore it can promise that the returned value will have any type, as long as it does not return at all. The same strategy can be used to implement contradiction : Void.

Therefore termination is a desirable property of functions, especially when they are used to model logic.

Universe stratification If Type is the type of types, what is the type of Type? It is well known that a system with Type : Type is inconsistent [Gir72; Hur95], by an argument analogous to Russell's paradox in naïve set theory.

A frequently used solution is introducing *universe levels*, where the type of types is Type_0 and the type of Type_i is, depending on the system, either Type_{i+1} or Type_j for any $j > i$ (which is called cumulativity).

However, for presentational purposes, a type system with Type : Type is often easier to explain, as universe level manipulation does not obscure the subject matter of the presented calculus. In such cases, it is implied that the universe hierarchy is independent from the presented ideas and the type universe can be stratified as necessary. This is also the case of this dissertation.

Other restrictions Strict positivity [CP90] is a requirement usually imposed on definitions of inductive type families, where recursive occurrences of the type family are required to be found only in strictly positive positions.

2.2.3 Propositional equality

We can also define a predicate that expresses that two values are the same.

$$\begin{aligned}&\textbf{data}\ (\equiv) : \{a : \mathsf{Type}\} \to a \to a \to \mathsf{Type}\ \textbf{where}\\ &\quad \mathsf{Refl} : \{x : a\} \to x \equiv x\end{aligned}$$

The name Refl stands for "reflexivity" and Refl $\{x\}$ is a proof that x is equal to x. The argument x of Refl is implicit and we will normally omit it.

Values of type $y \equiv z$ serve as evidence that y is the same value as z. If y and z were different, there would be no single value we could give to Refl's argument x – both Refl $\{\mathsf{x} = y\}$, which has the type $y \equiv y$, and Refl $\{\mathsf{x} = z\}$, which has the type $z \equiv z$, mismatch with $y \equiv z$ on one of the sides of $(\equiv)$ because $y \neq z$.

Definitional equality The above explicit notion of equality is called *propositional equality*. It allows us to "materialise" *definitional equality*, which is the implicit notion of equality present in the language, decided and applied transparently and automatically whenever the language needs to check that two types match.

Definitional equality is often defined as the reflexive transitive symmetric structural closure of reduction. An example of two terms that are definitionally equal in Idris could be $2 + 2$ and 4. Both have a common reduct, namely 4, and Idris will therefore let us use them interchangeably.

```
the (Vect 4 ℕ) (0 :: 1 :: 2 :: 3 :: Nil)          — typechecks
the (Vect (2 + 2) ℕ) (0 :: 1 :: 2 :: 3 :: Nil)    — typechecks
the (2 + 2 ≡ 4) (Refl {x = 2 + 2})                — typechecks
the (2 + 2 ≡ 4) (Refl {x = 4})                    — typechecks
the (2 + 2 ≡ 4) Refl                              — typechecks
```

2.2.3.1 Limits of definitional equality

However, there are values that we know are equal but which are not *definitionally* equal, and therefore are not freely interchangeable. An example would be n and $n + \mathsf{Z}$ in the following function.

```
strangeIdentity✗ : Vect n a → Vect (n + Z) a
strangeIdentity✗ xs = xs      — type mismatch: n vs. n + Z
```

Since $(+)$ is defined by pattern matching on its first argument, $n + \mathsf{Z}$ does not reduce to n (unlike $\mathsf{Z} + n$). Idris therefore cannot see that both expressions are the same.

We can, however, express their equality explicitly by defining a function that computes a value of type $n \equiv n + \mathsf{Z}$ for any given n.

```
cong : (f : a → b) → m ≡ n → f m ≡ f n
cong f Refl = Refl
```
(2.20)

```
nPlusZ : (n : ℕ) → n ≡ n + Z
nPlusZ   Z     =  Refl                 — Z = Z + Z holds definitionally
nPlusZ  (S k)  =  cong S (nPlusZ k)    — k = k + Z comes from IH
```

If n is zero, the equality holds definitionally. If n is a successor, we can apply nPlusZ recursively to obtain the inductive hypothesis, from which we can get the desired proof using cong. The function cong implements the fact that propositional equality is a congruence.

Now we can implement strangeIdentity and have it accepted by the type checker.

```
subst : (f : a → Type) → x ≡ y → f x → f y
subst f Refl = λz. z
```
(2.21)

```
strangeIdentity : Vect n a → Vect (n + Z) a
strangeIdentity {a} {n} xs = subst (λk. Vect k a) (nPlusZ n) xs
```

The function subst converts values from type $f\ x$ to type $f\ y$, provided that x is equal to y. We then use this function, together with a proof that $n \equiv n + \mathsf{Z}$, to convert xs, which has the type Vect $n\ a$, to the type Vect $(n + \mathsf{Z})\ a$.

In this example, propositional equality allowed us to compose isolated pieces of definitional equality into a "bigger picture" by enabling us to speak explicitly about it.

2.2.3.2 Pattern matching on Refl

In the above definitions of cong and subst, we made use of equality proofs by pattern matching on them. To explain how the definitions work, let us restate them, with

some implicits made explicit. We will build on the explanation given in Section 2.1.4.

$$\begin{aligned}&\mathsf{cong} : (f : a \to b) \to (m : a) \to (n : a) \to (eq : m \equiv n) \to f\ m \equiv f\ n\\&\mathsf{cong}\ f\ [x]\ [x]\ \left(\lceil \mathsf{Refl} \rceil\ \{\mathsf{x} = x\}\right) = \mathsf{Refl}\ \{\mathsf{x} = f\ x\}\end{aligned} \tag{2.22}$$

The only possible constructor of $(eq : m \equiv n)$ is Refl. We therefore make it forced using the forced-constructor brackets, $\lceil \cdot \rceil$, and we also give an explicit name to its implicit argument x. Then however, by type-correctness, both m and n must be equal to x because the type of $\mathsf{Refl}\ \{\mathsf{x} = x\}$ is $x \equiv x$. Finally, as explained in Section 2.1.8.2, since both m and n are forced to be the same as x, the required type of the RHS becomes $f\ x \equiv f\ x$, which is easily satisfied using Refl.

In the above clause, the variables m and n do not exist as two separate entities. Instead, they have been replaced with x everywhere, where x is the argument of Refl.

As shown in Sections 2.1.4.2 and 2.1.4.4, there are however several choices how to annotate forced patterns in the above definition of cong while keeping the same type signature.

$$\mathsf{cong}\ f\ m\ [m]\ \left(\lceil \mathsf{Refl} \rceil\ \{\mathsf{x} = [m]\}\right) = \mathsf{Refl}\ \{\mathsf{x} = f\ m\} \tag{2.23}$$

In the above clause, we reinterpret the match that by type correctness, both n and the argument x of Refl must be equal to m.

$$\mathsf{cong}\ f\ [n]\ n\ \left(\lceil \mathsf{Refl} \rceil\ \{\mathsf{x} = [n]\}\right) = \mathsf{Refl}\ \{\mathsf{x} = f\ n\} \tag{2.24}$$

Finally, we can also reinterpret the match to force m and x from n. The consequences of different interpretations are described in Section 2.1.4.4.

2.2.3.3 Erasure and propositional equality

In Listings 2.23 and 2.24, the whole value of eq, the proof of equality, is forced from other patterns. Therefore, the fourth argument of cong is unused and could be erased in both cases.

In Listing 2.22, we project x out of Refl to use it on the RHS. However, on the RHS, it will end up only as an argument to Refl again. If no other parts of the code use the argument of Refl, x will be unused, and the whole fourth argument of cong will be unused and could be erased.

In practice, equality proofs will be erasable because the argument of Refl will be obtainable/forceable from other parts of the pattern.

Token type target elimination However, Mishra-Linger observes [ML08, page 133] that unrestricted erasure of *token type target elimination*, which includes erasure of equality proofs, does not preserve strong normalisation and therefore should not be allowed. More details are given in Section 9.2.1.8.

2.2.3.4 Heterogeneous equality

We should be able to prove the following property of vector concatenation.

```
rightNeutral✗ : {xs : Vect n a} → xs ++ Nil ≡ xs   — type mismatch: n vs. n + Z
```

However, we cannot even *state* it – the above type signature will not typecheck. The problem is that (≡) can relate only values of the same type but Idris does not see Vect n a, which is the type of xs, and Vect $(n + \mathsf{Z})$ a, which is the type of xs ++ Nil, as equal – they are not definitionally equal. We therefore need to relax the requirements of (≡).

Also known as the John Major equality, heterogeneous equality [McB99] allows comparison of values of possibly different types.

```
data (≡') : a → b → Type where
  Refl' : x ≡' x
```

Now we can state the property we wanted.

```
rightNeutral : {xs : Vect n a} → xs ++ Nil ≡' xs
— implementation omitted
```

2.2.3.5 Decidable equality

For some types, equality is decidable. For any such type a, we can implement a decision procedure. (See Section 2.2.2.2 for the definition of Dec.)

```
decEq_a : (x : a) → (y : a) → Dec (x ≡ y)
```

2.2.4 Simulating mutual recursion

In a language without mutual recursion, we can still encode mutual recursion in various ways that are often sufficient.

2.2.4.1 In data types

Mutually recursive data types can be simulated by passing one data type as a parameter of the other, as shown below.

```
data Even' : (ℕ → Type) → ℕ → Type where
  EZ : EZ odd Z
  ES : odd n → Even' odd (S n)

data Odd : ℕ → Type where
  OB : Odd (S Z)
  OS : Even' Odd (S n) → Odd (S (S n))
```

```
Even : ℕ → Type
Even = Even' Odd
```

In languages like Agda, which requires programmers to explicitly distinguish between parameters and indices, the first argument to Even' must be marked as parameter, not as an index, to avoid strict positivity issues.

2.2.4.2 Using local let bindings

Some patterns of mutual recursion can be simulated if the language has local pattern-matching let bindings.

```
even : ℕ → Bool
even   Z    = True
even  (S n) =
  let
    odd : ℕ → Bool
    odd   Z    = False
    odd  (S n) = even n
  in odd n
```

The function even can call odd because odd is defined in a **let** block within its body, while the function odd can call even as a recursive call.

With clauses (Section 2.2.5) and all their variants (case expressions, rewrite clauses) are a naturally occurring example of this kind of recursion.

2.2.4.3 Using tagged dispatch

We can write a function that uses the first argument to determine which of the mutually recursive functions should be invoked, as shown in Figure 2.4. Because we are working in a dependently typed language, we can accurately describe all involved types.

2.2.4.4 Erasure-polymorphic tagged dispatch

In Chapter 5, I introduce a calculus that models erasure in the types. With that approach, the mutually recursive functions modelled in the previous section could have not only different types but also different erasure patterns, because the erasure pattern is a part of the type of a function. Then everything, including erasure *inference*, works as one would expect in mutually recursive functions.

2.2.5 With clauses

Sometimes we need to pattern match on values that are not arguments to the function being defined, for example in filter, whose one possible implementation is shown in Listing 2.14.

```
data Tag : Type where
  Even : Tag
  Odd  : Tag

funTy : Tag → Type
funTy Even = ℕ → Bool
funTy Odd  = ℕ → Bool

evenOdd : (tag : Tag) → funTy tag
= let
    even : ℕ → Bool
    even   Z    = True
    even (S n)  = evenOdd Odd n
    odd : ℕ → Bool
    odd    Z    = False
    odd  (S n)  = evenOdd Even n
in let
      dispatch : (tag : Tag) → funTy tag
      dispatch Even = even
      dispatch Odd  = odd
  in dispatch
```

FIGURE 2.4: Mutual recursion via tagged dispatch

Another possible implementation uses *with clauses*, introduced by McBride and McKinna in Epigram [MM04], which can be seen as an extension of pattern guards [EJ00].

```
filter : (p : a → Bool) → List a → List a
filter p Nil = Nil
filter p (x :: xs) with p x                    (2.25)
filter p (x :: xs) | True  = x :: filter p xs
filter p (x :: xs) | False = filter p xs
```

A with clause effectively adds another pattern to the LHS of its parent pattern matching clause, but the newly added pattern can still inform patterns to the left of it.

An example would involve decidable equality. Let $\mathsf{decEq}_{\mathbb{N}}$ decide equality of natural numbers.

```
decEqℕ : (x : ℕ) → (y : ℕ) → Dec (x ≡ y)
```

The function $\mathsf{decEq}_{\mathbb{N}}$ returns either Yes with a proof of equality or No with a proof of nonequality. We can then use it as shown in the following (slightly contrived)

function.

```
semiCong : (f : ℕ → a) → (x : ℕ) → (y : ℕ) → Maybe (f x ≡ f y)
semiCong f x y with decEq_ℕ x y
semiCong f x [x] | Yes Refl    = Just Refl
semiCong f x y   | No contra   = Nothing
```
(2.26)

The function semiCong returns a proof that $f\ x \equiv f\ y$ whenever it turns out that x and y are equal. In the case with Yes Refl, the pattern for y has been forced to $[x]$ as a result of matching in the newly added pattern.

With clauses are supported at least in Epigram, Idris, and Agda.

Abbreviation If the subclause of with clause has the same LHS as the parent clause, the LHS need not be repeated. Agda allows replacing the repeated LHS with "..." and Idris allows a complete elision of the LHS, yielding the following definition of filter.

```
filter : (p : a → Bool) → List a → List a
filter p Nil = Nil
filter p (x :: xs) with p x
  | True  = x :: filter p xs
  | False = filter p xs
```

2.2.5.1 Implementation

With clauses are implemented using an auxiliary function with the extra argument. The definition of filter in Listing 2.26 desugars into the following.

```
filter : (p : a → Bool) → List a → List a
filter p Nil = Nil
filter p (x :: xs) =
  let
    filter' : (p : a → Bool) → a → List a → Bool → List a
    filter' p x xs True  = x :: filter p xs
    filter' p x xs False = filter p xs
  in filter' p x xs (p x)
```
(2.27)

In general, the arguments of the auxiliary function are all the pattern variables, plus the newly introduced argument. [MM04]

Mutual recursion and termination In Listing 2.27, the auxiliary function is defined in a let binding, as opposed to a separate top-level function. This has two advantages.

- The language does not need mutual recursion. The function filter is allowed to call filter' because filter' is defined in a let binding. However, filter' is allowed to

call filter because it is itself embedded in the body of filter and the call is just a recursive call.

- Termination may be easier to prove. While in terms of termination checking, the function shown in Listing 2.27 is not much different from two mutually recursive functions, Section 2.2.5.4 shows that we can do better.

2.2.5.2 High-level presentation

In non-desugared code, Idris allows calling the auxiliary function directly. This is very useful for programming with recursive views and related structures, as we will see in later sections of this chapter. For now, we will demonstrate the feature with the following definition.

$$
\begin{array}{l}
\mathsf{reverse} : \mathsf{List}\ a \longrightarrow \mathsf{List}\ a \\
\mathsf{reverse}\quad xs\quad \textbf{with}\ \mathsf{Nil} \\
\mathsf{reverse}\quad \mathsf{Nil}\quad \mid acc\ =\ acc \\
\mathsf{reverse}\ \ (y :: ys)\ \mid acc\ =\ \underbrace{\mathsf{reverse}\ ys \mid (y :: acc)}_{\text{recursive call}}
\end{array}
$$

The example above shows how to reverse a list with an accumulator but no explicit auxiliary function. The extra argument introduced in the with clause is the accumulator. It starts off empty and the clause for $(y :: ys)$ calls the auxiliary function directly, using the symbol | to pass it $(y :: acc)$ explicitly as the extra argument.

In Agda, if reduction of the outer function succeeds but the reduction of the auxiliary function is blocked, Agda will print the half-reduced value with similar syntax involving the symbol |. For example, if $p\ x$ does not reduce, then given the definition of filter in Listing 2.25, Agda will print the half-reduced value of filter $p\ (x :: xs)$ as filter $p\ (x :: xs) \mid p\ x$. This can also be interpreted as an application of the auxiliary function to all its arguments, including to $p\ x$ for the added argument.

2.2.5.3 Rewriting

Sometimes, we need to *rewrite* a complex expression in types to another expression that we know is equal to the original one. Suppose we need to implement the following function.

$$\mathsf{pf} : (p + q) \equiv (r + s) \longrightarrow \mathsf{Vect}\ (p + q)\ a \longrightarrow \mathsf{Vect}\ (r + s)\ a$$

Since both lengths are equal, we should be able to just return the provided vector.

However, we cannot simply match on the proof of equality because there is no way to choose the argument to Refl such that it is definitionally equal to both $(p + q)$

and $(r + s)$, not even with forced patterns[4].

```
pf (Refl {x = ???}) = ...    — type mismatch on the LHS
```

We can however introduce an auxiliary function with an extra variable that will mediate the equality. We arbitrarily pick one of the two expressions, for example $(p + q)$, and represent it with a variable, which we will name pq. We will then write the type signature of the auxiliary function, where all occurrences of $(p + q)$ have been replaced with pq. Since pq is a variable, it can now be forced to $[r + s]$ by pattern matching on Refl in the auxiliary function.

```
pf : (p : ℕ) → (q : ℕ) → (r : ℕ) → (s : ℕ)
     → (p + q) ≡ (r + s) → Vect (p + q) ℕ → Vect (r + s) ℕ
pf p q r s =
  let                                                          (2.28)
    pf' : (pq : ℕ) → pq ≡ (r + s) → Vect pq ℕ → Vect (r + s) ℕ
    pf' [r + s] Refl = λxs. xs
  in pf' (p + q)
```

Generalisation We can interpret the type signature of pf′ as a generalisation of the type signature of pf over subterms $p + q$. In the calling function, pf, the specialisation of pf′ with $pq = p + q$ gives us a value of exactly the type that we need to implement pf. In the called function, pf′, the generalised form of the type gives us the ability to pattern match on Refl, while specialising to $pq = r + s$.

Change of perspective We might also say that in the caller, $pq = p + q$ but in the callee, $pq = r + s$. This change of perspective is a very useful technique: one can easily prove statements that are definitionally true in the caller, where we know that $pq = p + q$, and then pass the corresponding proof terms to the callee. In the callee, we can use additional definitional equalities coming from knowing that $pq = r + s$, plus the proofs passed on from the caller (which may no longer hold definitionally). Among other places, this is used for the inspect idiom (see below).

This also gives us the ability to deal with the "green slime" [McB14c] in constructor indices[5] – if we generalise over all conflicting values to replace them with variables, we become able to match on constructors with "green slime", forcing the generalised variables to the appropriate function applications.

"Magic" with The above is more conveniently achieved using "magic with" [Agd17b]. In Section 2.2.5.1, I described a way to generate auxiliary functions that implement the *with* idiom. We can extend the desugaring step by generating a more general

[4]See Section 2.1.8.2 for an explanation how pattern clauses are checked.

[5]In Agda, types whose constructors have functions in indices (coloured green by default) often make Agda refuse to case-split on values of these types because Agda cannot decide which constructors of the type family to include in the given split.

type signature for the auxiliary function by automating what we did manually in Listing 2.28 – by replacing all occurrences of the with-inspected expression in the type signature for the newly added variable. This implementation of **with** lets us write the following definition.

$$\begin{aligned}
&\mathsf{pf} : (p : \mathbb{N}) \to (q : \mathbb{N}) \to (r : \mathbb{N}) \to (s : \mathbb{N}) \\
&\qquad \to (p + q) \equiv (r + s) \to \mathsf{Vect}\ (p + q)\ \mathbb{N} \to \mathsf{Vect}\ (r + s)\ \mathbb{N} \\
&\mathsf{pf}\ \ p\ \ q\ \ r\ \ s\ \ \ eq\ \ \ xs\ \ \textbf{with}\ p + q \\
&\mathsf{pf}\ \ p\ \ q\ \ r\ \ s\ \ \mathsf{Refl}\ \ xs\ \mid [r + s]\ =\ xs
\end{aligned} \tag{2.29}$$

This definition reads somewhat unusually – we inspect $(p + q)$ to learn that it is forced to be $[r + s]$ – but the underlying principles explain how the definition works.

We also had to η-expand the definition slightly. The definition in Listing 2.28 could invoke the auxiliary function as $\mathsf{pf}'(p + q)$ because we inserted the newly added argument pq at the beginning of the type signature. However, mechanical desugaring of with clauses usually inserts the newly added argument at the end, which forces us to η-expand the definition to obtain the one found in Listing 2.29.

"Magic with" makes the programming language acknowledge that the inspected value is equal to the pattern, which allows us to write the following definition.

$$\begin{aligned}
&\mathsf{filtLem} : \big(q_1 : F\ (\mathsf{filter}\ p\ xs)\big) \to \big(q_2 : F\ (x :: \mathsf{filter}\ p\ xs)\big) \to F\ \big(\mathsf{filter}\ p\ (x :: xs)\big) \\
&\mathsf{filtLem}\ \ \{p\}\ \{x\}\ q_1\ q_2\ \ \textbf{with}\ p\ x \\
&\mathsf{filtLem}\ \ \{p\}\ \{x\}\ q_1\ q_2\ \mid \mathsf{False}\ =\ q_1 \\
&\mathsf{filtLem}\ \ \{p\}\ \{x\}\ q_1\ q_2\ \mid \mathsf{True}\ \ =\ q_2
\end{aligned}$$

In the clause for False, we know that filter $p\ (x :: xs)$ reduces to filter $p\ xs$, and thanks to "magic with", this equality is definitional. This allows us to use q_1 on the RHS.

In the clause for True, we know that filter $p\ (x :: xs)$ reduces to $x ::$ filter $p\ xs$, which allows us to use q_2 as the return value.

The inspect idiom When generalising the type signature for the auxiliary function, we replace all subterms that match the with-inspected expression. However, sometimes we may need an explicit proof that the with-inspected expression equals the corresponding pattern, which is especially useful if:

- the generalisation mechanism fails to spot subterms that are not syntactically identical but equivalent to the rewritten expression;
- we get additional occurrences of the rewritten expression from further reduction *after* generalisation has taken place;
- we **with**-inspect pairs/tuples of expressions or similar values, which blocks generalisation on the components of the pair/tuple entirely.

We can obtain an explicit proof of equality by creating an auxiliary function with *two* extra parameters rather than one. Let us consider the following (slightly contrived)

function that introduces the sugared syntax.

```
f : (p : a → Bool) → (x : a) → Either (p x ≡ True) (p x ≡ False)
f p x with p x proof eq
f p x | True  = Left eq
f p x | False = Right eq
```

The extra argument is the proof *eq*. In the clause for True, the type of *pf* is $p\ x \equiv \mathsf{True}$. In the clause for False, its type is $p\ x \equiv \mathsf{False}$. The above definition desugars as follows.

```
f : (p : a → Bool) → (x : a) → Either (p x ≡ True) (p x ≡ False)
f p x =
  let
    f' : (p : a → Bool) → (x : a)
         → (px : Bool) → p x ≡ px
         → Either (px ≡ True) (px ≡ False)
    f' p x True  eq = Left eq
    f' p x False eq = Right eq
  in f' (p x) Refl
```

Besides the usual extra argument *px*, we added a proof that $p\ x \equiv px$. In the invocation of f', we pass Refl there.

This is known as the *inspect* idiom [Agd17a]. In Agda, it is implemented in the standard library using "magic with", without other compiler support.

Rewrite clauses The trick in Listing 2.29 is so useful that it gets its own syntax in Agda and Idris. The syntax uses the keyword **rewrite** and in Agda, it simplifies the program as follows.

```
pf : (p + q) ≡ (r + s) → Vect (p + q) ℕ → Vect (r + s) ℕ
pf eq rewrite eq = λxs. xs
```
(2.30)

If $eq : X \equiv Y$ then a pattern clause of the form

```
f p1 ... pn rewrite eq = R
```

is syntax sugar for the following. Note the similarity with Listing 2.29.

```
f p1 ... pn with eq with X
f p1 ... pn | Refl | [Y] = R
```

In Agda, rewrite clauses are closely related to with clauses and can be mixed with them. In Idris, the **rewrite** sugar works at the expression level. In this dissertation, I will use the Agda variant.

Our function pf has thus shrunk significantly from its initial fully explicit implementation in Listing 2.28, through an implementation with "magic with" in Listing 2.29 to the one-liner in Listing 2.30.

2.2.5.4 Case expressions

Like with clauses, case expressions can be desugared to local let-bound pattern matching definitions as well. The difference between case expressions and with clauses is that case expressions are *expressions* and thus can appear anywhere in an expression context. The consequences of this are described below.

There are two different approaches – a minimalistic one and a maximalistic one. Let us first look at an example.

```
filter : (a → Bool) → List a → List a
filter p    Nil    = Nil
filter p (x :: xs) =
  case p x of
    True  ⇒ x :: filter p xs
    False ⇒ filter p xs
```

The minimalistic approach desugars a case expression to a local function with just one argument – the scrutinee.

```
filter : (a → Bool) → List a → List a
filter p    Nil    = Nil
filter p (x :: xs) =
  let
    f : Bool → List a
    f  True  = x :: filter p xs
    f  False = filter p xs
  in f (p x)
```
(2.31)

The maximalistic approach includes all pattern variables in scope at point of the case expression.

```
filter : (a → Bool) → List a → List a
filter p    Nil    = Nil
filter p (x :: xs) =
  let
    f : (a → Bool) → a → List a → Bool → List a
    f  p  x  xs  True  = x :: filter p xs
    f  p  x  xs  False = filter p xs
  in f p x xs (p x)
```
(2.32)

This maximalistic approach, combined with good type inference and elaboration, can provide basic forms of dependent pattern matching because the auxiliary function has all the necessary components – in fact, it is identical to a function generated by a with clause.

However, the main problem in desugaring of case expressions is determining the right type signature for f, since the surface syntax of case expressions normally does not include type annotations and, unlike with clauses, case expressions can occur deep in expressions on the RHS, where the appropriate return type is no longer clear. Augustsson notes that this problem is undecidable and Cayenne requires explicit type annotations on case expressions [Aug99], while Idris requires that the type annotation is "inferrable", which is defined by the implementation. Coq's **match**, mentioned in Section 2.1.8.2, allows explicit type annotations.

Magic case Like with "magic with" (Section 2.2.5.3), the desugaring procedure could also generalise the scrutinee in the type of the auxiliary function for case expressions, yielding "magic case".

Termination checking In Section 2.2.5.1, I mentioned that using local let binding for auxiliary definitions may make it easier to prove termination for recursive functions than mutually recursive auxiliary definitions.

The "maximalistic" definition of filter in Listing 2.32 is not very different from a definition with a mutually recursive auxiliary function. However, the minimalistic definition in Listing 2.31 invokes filter recursively on the list xs, which is bound as an argument of filter, not as an argument of the auxiliary function and is therefore structurally smaller than $x :: xs$.

Direct access to the pattern variables of the parent function may make this approach terminate more obviously, even without more sophisticated approaches like the size-change principle [LJBA01].

2.2.6 Well-founded recursion

To keep termination checking tractable, dependently typed languages allow only recursive calls with *decreasing* arguments, where "decreasing" means the notion of "being a syntactic subterm". However, there are other non-syntactic and useful notions of "decreasing" that we would like to use.

A good example is (functional) Quicksort. We *know* that the function terminates but the computer cannot see it because the arguments to the recursive calls have a more complicated relationship with $(x :: xs)$ than being its subterms.

$$
\begin{aligned}
&\mathsf{qsort} : \mathsf{List}\ \mathbb{N} \longrightarrow \mathsf{List}\ \mathbb{N} \\
&\mathsf{qsort}\quad \mathsf{Nil}\quad = \mathsf{Nil} \\
&\mathsf{qsort}\ \ (x :: xs)\ = \mathsf{qsort}\ (\mathsf{filter}\ (\leq x)\ xs) +\!\!+ x :: \mathsf{qsort}\ (\mathsf{filter}\ (> x)\ xs)
\end{aligned}
$$

This is a case where well-founded recursion can help.

Accessibility We say that an element x of type a is *accessible* by relation $<$ if all elements y such that $y < x$ are accessible. The notion of accessibility does not define any base case explicitly ("accessible from where?"), but there is still a natural base case where there are *no* elements smaller than x.

$$\begin{aligned}&\textbf{data}\ \mathsf{Acc} : (a \to a \to \mathsf{Type}) \to a \to \mathsf{Type}\ \textbf{where}\\&\quad \mathsf{MkAcc} : \left(acc : (y : a) \to (y < x) \to \mathsf{Acc}\ (<)\ y\right) \to \mathsf{Acc}\ (<)\ x\end{aligned}$$

Well-founded relations Relation $(<) : a \to a \to \mathsf{Type}$ is well-founded if all elements in a are accessible with $<$.

$$\begin{aligned}&\mathsf{WellFounded} : (a \to a \to \mathsf{Type}) \to \mathsf{Type}\\&\mathsf{WellFounded}\ (<) = (x : a) \to \mathsf{Acc}\ (<)\ x\end{aligned}$$

A (constructive) proof of well-foundedness is therefore a function that gives us a proof of accessibility for any desired element.

2.2.6.1 Constructing proofs of accessibility

Let us give a proof of well-foundedness of $<$ on natural numbers. Since we already use the symbol $<$ for the Boolean-valued function, we will use $\prec$ for the type-level relation.

First, we need to define the relation $\prec$. We will use the formulation from the standard library of Idris and Agda. Although accessibility is easier to prove for an alternative formulation with $\mathsf{LEZ} : n \preceq n$ and $\mathsf{LES} : m \preceq n \to m \preceq \mathsf{S}\ n$, this formulation seems to be more useful in general.

$$\begin{aligned}&\textbf{data}\ (\preceq) : \mathbb{N} \to \mathbb{N} \to \mathsf{Type}\ \textbf{where}\\&\quad \mathsf{LEZ}\ :\ \mathsf{Z} \preceq n\\&\quad \mathsf{LES}\ :\ m \preceq n \to \mathsf{S}\ m \preceq \mathsf{S}\ n\end{aligned}$$

$$\begin{aligned}&(\prec) : \mathbb{N} \to \mathbb{N} \to \mathsf{Type}\\&m \prec n = \mathsf{S}\ m \preceq n\end{aligned}$$

As a matter of convention, "LE" means "less than or equal" and "LT" means "strictly less than".

Now we can give a proof that $\prec$ is well founded on the naturals.

$$\begin{aligned}&\mathsf{trans}_{\preceq} : x \preceq y \to y \preceq z \to x \preceq z\\&\mathsf{trans}_{\preceq}\ \ \mathsf{LEZ}\ \ \ \ \ \ \ \ \ \ \ \ \_\ \ \ \ \ \ \ \ \ \ \ = \mathsf{LEZ}\\&\mathsf{trans}_{\preceq}\ \ (\mathsf{LES}\ xLEy)\ \ (\mathsf{LES}\ yLEz)\ \ = \mathsf{LES}\ (\mathsf{trans}_{\preceq}\ xLEy\ yLEz)\end{aligned}$$

```
wf≺ : WellFounded (≺)
wf≺ x = MkAcc (acc x)
  where

    acc : (x : ℕ) → (y : ℕ) → y ≺ x → Acc (≺) y
    acc (⌈S⌉ x′) y (⌈LES⌉ yLEx′)
        = MkAcc (λz. λzLTy.
            acc x′ z (trans≤ zLTy yLEx′))
```

We give a proof of accessibility of any specific x by first generalising over x using a function acc. By pattern matching on the proof that $y \prec x$, we learn that it must have been constructed using LES because $y \prec x$ is an alias for S $y \leq x$ and LEZ cannot be used for proofs with S _ on the LHS.

From that, we also know that x is a successor, namely S x'. This will allow us to invoke acc recursively later.

In the lambda inside MkAcc, we have to prove accessibility of any z that we know is strictly smaller than y. Because we know that $y \leq x'$, we know that $z \prec x'$ by transitivity and we can invoke acc on x' and z recursively because x' is a direct subterm of x.

Sized types We can easily construct accessibility predicates for any *size* mapping, which is a map from a type to $\mathbb{N}$, using exactly the same approach as we used with $\prec$ and the natural numbers.

```
(⊏s) : {s : a → ℕ} → a → a → Type
x ⊏s y = s x ≺ s y


wf⊏ : (s : a → ℕ) → WellFounded (⊏s)
wf⊏ s x = MkAcc (acc (s x))
  where

    acc : (sx : ℕ) → (y : a) → s y ≺ sx → Acc (⊏s) y
    acc (⌈S⌉ sx′) y (⌈LES ⌉syLEsx′)
        = MkAcc (λz. λszLTsy.
            acc sx′ z (trans≤ szLTsy syLEsx′))
```

This is very useful because we immediately get a proof of well-foundedness of $\sqsubset_{\mathsf{length}}$ on lists, $\sqsubset_{\mathsf{depth}}$ on trees etc.

The standard library of Agda contains a more general proof for any preimage.

2.2.6.2 Example: Quicksort

We can now implement functional Quicksort using well-founded recursion. First, we need to prove that filter does not make lists longer.

```
leS : m ≤ n → m ≤ S n
leS  LEZ     = LEZ
leS  (LES x) = LES (leS x)
```

```
filterLen : {p : a → Bool} → {xs : List a} → length (filter p xs) ≤ length xs
filterLen {xs = Nil} = LEZ
filterLen {p} {xs = x :: xs} with p x                                    (2.33)
  | True  = LES filterLen
  | False = leS filterLen
```

The definition of filterLen uses "magic with" (Section 2.2.5.3) to reduce the application of filter to a more specific value in each branch.

Now we can proceed to define qsort.

```
qsort : List ℕ → List ℕ
qsort     xs     with wf⊏ length xs
qsort     Nil    | _                = Nil
qsort  (x :: xs) | ⌈MkAcc⌉ acc  =
    (qsort (filter (≤ x) xs) | acc _ (LES filterLen))
    ++ (x :: Nil)
    ++ (qsort (filter (> x) xs) | acc _ (LES filterLen))
```

This definition uses explicit invocation of auxiliary **with** functions using the symbol |, as described in Section 2.2.5.2.

The key trick in this definition of qsort is that although we still recurse on lists that are not syntactically smaller, the application of *acc* in the recursive call is syntactically smaller than MkAcc *acc* and Idris accepts the function as structurally recursive on the accessibility proof.

2.2.6.3 Example: Mergesort

In Section 2.1.4, we already saw a data type representing splits of lists.

```
data Split✗ : List a → Type where
  MkSplit✗ : (ls : List a) → (rs : List a) → Split✗ (ls ++ rs)
```

Constructor MkSplit✗ contains *ls*, the left part of the list, and *rs*, the right part of the list, and it represents the list *ls* ++ *rs*, where ++ is the list concatenation operator.

```
pushL : (x : a) → Split xs → Split (x :: xs)
pushL  x  SNil                = SOne x
pushL  x  (SOne y)            = SMore x Nil y Nil
pushL  x  (SMore y ys z zs)   = SMore x (y :: ys) z zs

halve : (xs : List a) → Split xs
halve                Nil  = SNil
halve          (x :: Nil) = SOne x
halve      (x :: y :: xs) with 1 + length xs
halve      (x :: y :: xs) |       Z  = SMore x Nil y xs
halve      (x :: y :: xs) |   S Z  = SMore x Nil y xs
halve      (x :: y :: Nil) | S (S k) = SMore x Nil y Nil
halve  (x :: y :: z :: xs) | S (S k) = pushL x (halve (y :: z :: xs) | k)
```

FIGURE 2.5: Halving a list, producing a Split as shown in Listing 2.34

We could therefore write a function halve that takes a list and returns a Split of it containing the two halves of the list.

```
halve : (xs : List a) → Split xs
```

The fact that halve splits lists into (approximately) equal parts is not encoded in the return type but that is not necessary for our purposes.

However, we do need to make it explicit that the components of a split are shorter than the whole list – that we we are not going to split the list xs into $\mathsf{MkSplit}_{\not x}$ Nil xs, for example. Let us therefore use a different type for list splits.

```
data Split : List a → Type where
  SNil  : Split Nil
  SOne  : (x : a) → Split (x :: Nil)
  SMore : (x : a) → (xs : List a)
        → (y : a) → (ys : List a)
        → Split (x :: xs ++ y :: ys)
```

(2.34)

This time, the constructors represent empty lists, singleton lists, and lists that split into two nonempty parts.

We can now implement the halving function, which can be seen in Figure 2.5, and prove two lemmas about lists (Figure 2.6).

```
refl≤ : n ≤ n
refl≤ {x = Z}   = LEZ
refl≤ {x = S n} = LES refl≤

shorterL : xs ⊏length xs ++ y :: ys
shorterL {xs = Nil}    = LES LEZ
shorterL {xs = x :: xs} = LES shorterL

shorterR : ys ⊏length x :: xs ++ ys
shorterR {xs = Nil}    = LES refl≤
shorterR {xs = x :: xs} = leS (shorterR {x = x})
```

FIGURE 2.6: Two lemmas about lists

The function that merges two sorted lists, merge, is nothing surprising, either.

```
merge : List ℕ → List ℕ → List ℕ
merge Nil       ys      = ys
merge xs        Nil     = xs
merge (x :: xs) (y :: ys) with x ≤ y
  | True  = x :: merge xs (y :: ys)
  | False = y :: merge (x :: xs) ys
```
(2.35)

Finally, we can implement msort using everything that we have defined so far.

```
msort : List ℕ → List ℕ
msort xs          with wf⊏ length xs
msort xs          | _              with halve xs
msort []          | _              | SNil          = Nil
msort [x]         | _              | SOne x        = x :: Nil
msort (y :: ys ++ z :: zs) | ⌈MkAcc⌉ acc | SMore y ys z zs
  = merge
      (msort (y :: ys) | acc _ (shorterL {xs = y :: ys}))
      (msort (z :: zs) | acc _ (shorterR {ys = z :: zs}))
```
(2.36)

Like in the case with Quicksort, the termination checker observes that *acc* is a strictly decreasing argument and therefore msort is accepted as total.

2.2.6.4 Accessibility and strong induction

Well-founded recursion is to structural recursion what strong induction is to ordinary (weak) induction. Defining accessibility therefore corresponds to proving the strong induction principle from the weak induction principle. The power of well-founded recursion thus comes from a stronger induction hypothesis.

2.2.7 Domain predicates

Bove and Capretta [BC05] present a way of extracting a termination argument from the definition of a function, allowing it to be proven separately. Termination arguments are implemented by *domain predicates*, which describe which values of the input type are in the domain of the function – for which inputs the function is defined. A domain predicate is implemented as an accessibility predicate (Section 2.2.6) tailored for the function in question.

Bove-Capretta's method does not make proving termination *easier* per se; constructing the proof that all values belong to the domain of a function is exactly as hard as proving termination of the function itself. So although we may still have to resort to well-founded recursion when proving the termination argument, the proof is written separately and does not interfere with the business logic of the program we are writing and does not clutter its code.

A separate and explicit termination argument also makes it straightforward to *postulate* termination.

Besides separation of concerns, Bove-Capretta's method allows modelling partiality as well – in such cases, the domain predicate does not cover all possible inputs to the function.

2.2.7.1 Example: Quicksort

As an example, let us implement functional Quicksort again. First, we define a specialised accessibility/domain predicate.

```
data QSortAcc : List ℕ → Type where
  QNil  : QSortAcc Nil
  QCons : QSortAcc (filter (≤ x) xs)
          → QSortAcc (filter (> x) xs)
          → QSortAcc (x :: xs)
```

This allows us to define qsort' in a very straightforward way because each constructor of QSortAcc corresponds to exactly one pattern clause and it contains exactly the recursive fields needed for recursive calls in qsort'.

```
qsort' : (xs : List ℕ) → QSortAcc xs → List ℕ
qsort'  Nil       [QNil]                = Nil
qsort' (x :: xs) (⌈QCons⌉ accL accR) =
  qsort' (filter (≤ x) xs) accL
  ++ (x :: Nil)
  ++ qsort' (filter (> x) xs) accR
```

The definition of qsort' is not very different from the ordinary definition shown in Listing 2.15. There is one extra argument, which is just mechanically matched and

passed to recursive calls; nothing non-trivial happens with it.

The question of termination has been entirely deferred to the accessibility predicate: all recursive calls are structural because the structure of the accessibility predicate mirrors the call structure of our function.

Also note that all components of all accessibility proofs are either forced or end up as accessibility proofs in recursive calls.

Proving accessibility If we want to show that the function terminates on all inputs, we have to prove the domain predicate for any given value of the input type. In the case of Quicksort, we have to implement the following function.

```
qsortAcc : (xs : List ℕ) → QSortAcc xs
```

We can do it in any way we like but possibly the easiest way is to use general-purpose accessibility and reuse filterLen from Listing 2.33 in Section 2.2.6.2.

```
qsortAcc : (xs : List ℕ) → QSortAcc xs
qsortAcc     xs       with wf_⊏ length xs
qsortAcc     Nil      | _                = QNil
qsortAcc  (x :: xs)   | ⌈MkAcc⌉ acc  =
  QCons
    (qsortAcc _ | acc _ (LES filterLen))
    (qsortAcc _ | acc _ (LES filterLen))
```

The above proves that Quicksort terminates on all inputs, which allows us to write a total, terminating qsort, yet again.

```
qsort : List ℕ → List ℕ
qsort xs = qsort' xs (qsortAcc xs)
```

2.2.7.2 Example: Mergesort

The domain predicate for Mergesort is similar but unlike in Quicksort, we branch on a (non-recursive) view of the list, as seen in Listing 2.36. This intermediate step in computation splits our domain predicate into two mutually recursive definitions (Figure 2.7).
Here, the function halve comes from Figure 2.5.

If we wish to avoid mutual recursion, we could inline the function into the data type, obtaining the definition shown in Figure 2.8. Both versions lead to the same code, except for minor variation in type signatures.

```
mutual
  data MSAcc' : Split xs → Type where
    MSNil  : MSAcc' SNil
    MSOne  : MSAcc' (SOne x)
    MSMore : MSAcc (x :: xs)
             → MSAcc (y :: ys)
             → MSAcc' (SMore x xs y ys)

  MSAcc : List a → Type
  MSAcc xs = MSAcc' (halve xs)
```

FIGURE 2.7: Mutually recursive domain predicate for Mergesort

```
data MSAcc : Split xs → Type where
  MSNil  : MSAcc SNil
  MSOne  : MSAcc (SOne x)
  MSMore : MSAcc (halve (x :: xs))
           → MSAcc (halve (y :: ys))
           → MSAcc (SMore x xs y ys)
```

FIGURE 2.8: Non-mutually-recursive domain predicate for Mergesort

With the mutually recursive formulation of the domain predicate, we can implement msort' as follows.

```
msort' : (xs : List ℕ) → MSAcc xs → List ℕ
msort'      xs          acc                  with halve xs
msort'      [Nil]       [MSNil]              | SNil          = Nil
msort'    [x :: Nil]    [MSOne]              | SOne x        = x :: Nil
msort' [y :: ys ++ z :: zs] (⌈MSMore⌉ accL accR) | SMore y ys z zs
  = merge
      (msort' (y :: ys) accL)
      (msort' (z :: zs) accR)
```

Operationally, this definition splits its input argument, the list, into halves, and then it matches on the possible splits while forcing the list into the forms implied by each split. In this case, even though the list argument is fully forced in most clauses, it is still inspected – we constructed the halves from it.

The domain predicate can be shown to hold for any list.

```
msAcc : (xs : List ℕ) → MSAcc xs
msAcc        xs          with wf_⊏ length xs
msAcc        xs          | acc              with halve xs
msAcc        []          | acc              | SNil           = MSNil
msAcc      [x :: []]     | acc              | SOne x         = MSOne
msAcc  [y :: ys ++ z :: zs] | ⌈MkAcc⌉ acc   | SMore y ys z zs
  = MSMore
      (msAcc _ | acc _ (shorterL {xs = y :: ys}))
      (msAcc _ | acc _ (shorterR {ys = z :: zs}))
```

The functions shorterL and shorterR show that the components of a split are strictly shorter than the whole list. They are defined in Figure 2.6.

This finally leads to the implementation of msort, which puts all the pieces together.

```
msort : List ℕ → List ℕ
msort xs = msort' xs (msAcc xs)
```

2.2.8 Views

A view [Wad87b; MM04] of a structure is an alternative representation of the structure with the same meaning. Using an alternative representation of data can make programs clearer, easier to write, or more efficient (or all of the above).

This section makes heavy use of the notation introduced in Section 2.2.5.

2.2.8.1 Example: Last-element view of lists

Lists are defined as either being empty or having a head (left-most element) and a tail (the list without the left-most element). It is not therefore possible to match on the list from the *right*, which we could use to reverse lists.

```
reverse✗ : List a → List a
reverse✗ Nil          = Nil
reverse✗ (xs ++ x :: Nil) = x :: reverse✗ xs   — error: not a pattern: (xs ++ x :: Nil)
```

(2.37)

Although the LHS of the second clause looks unambiguous to humans, the symbol ++ is a user-defined function and as such it cannot be inverted. (It is common to define ++ and :: as right-associative at the same precedence level so $xs ++ x :: \mathsf{Nil} = xs ++ (x :: \mathsf{Nil})$.)

However, we can make a view of the list, indexed with the original list to keep the link between the original and the view.

```
data SnocView : List a → Type where
  SNil  : SnocView Nil
  SSnoc : (xs : List a) → (x : a) → SnocView (xs ++ x :: Nil)
```

The name "snoc" is the reverse of the word "cons", which is a traditional name for the operation that prepends an element at the beginning of a list. Linked lists that extend to the right by appending elements are therefore often called snoc-lists.

We can add a *covering function* that constructs the view from any list.

```
snocView : (xs : List a) → SnocView xs
snocView Nil           = SNil
snocView (x :: xs)     with snocView xs
snocView (x :: [Nil])  | SNil       = SSnoc Nil x
snocView (x :: [ys ++ y :: Nil]) | SSnoc ys y = SSnoc (x :: ys) y
```

The covering function itself uses the view recursively on the tail of the list, constructing the view of the whole list from the view of the tail.

Now we can implement reverse using the Snoc view.

```
reverse : List a → List a
reverse xs          with snocView xs
reverse [Nil]       | SNil       = Nil
reverse [xs ++ x :: Nil] | SSnoc xs x = x :: reverse xs
```
(2.38)

Notice how the function looks similar to the failed attempt in Listing 2.37. However, it works differently – it branches on the view and also projects the variables x and xs from the view, which forces the form of the whole list.

2.2.8.2 Recursive views

The definition of reverse in Listing 2.38 has two major drawbacks:

- it does not run in linear time because it calculates snocView of the tail in every recursive step;
- it is not accepted as terminating because the recursive call is not performed on an (obvious) subterm of the argument of the parent call.

One way of solving these problems is making SnocView recursive – instead of containing the prefix xs as a list, the constructor SSnoc could contain the prefix already in the

```
appAssoc : (xs : List a) → (ys : List a) → (zs : List a)
     → (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
appAssoc     Nil     ys  zs  = Refl
appAssoc  (x :: xs)  ys  zs  = cong (x ::) (appAssoc xs ys zs)

appNil : (xs : List a) → xs = xs ++ Nil
appNil     Nil     = Refl
appNil  (x :: xs)  = cong (x ::) (appNil xs)


snocViewRec : (xs : List a) → SnocViewRec xs
snocViewRec = svRec SRNil
 where
   svRec : SnocViewRec xs → (ys : List a) → SnocViewRec (xs ++ ys)
   svRec  {xs}  sxs     Nil     rewrite appNil xs
        = sxs
   svRec  {xs}  sxs  (y :: ys)  rewrite appAssoc xs (y :: Nil) ys
        = svRec (SRSnoc sxs y) ys
```

The function snocViewRec is implemented via its generalised form svRec, which describes a linear list traversal: at the beginning, the SnocViewRec is empty and *ys* is equal to the whole list. Each recursive step then moves one element from the beginning of *ys* to the end of *xs* (and thus *sxs*). The invariant that the whole list is equal to *xs* ++ *ys* is thus always kept.

FIGURE 2.9: Implementation of snocViewRec

form of a (recursive) snoc view. This gives rise to SnocViewRec.

```
data SnocViewRec : List a → Type where
  SRNil  : SnocViewRec Nil                                                    (2.39)
  SRSnoc : (sxs : SnocViewRec xs) → (x : a) → SnocViewRec (xs ++ x :: Nil)
```

In the definition above, *xs* becomes an implicit and the focus shifts to *sxs*, which is the SnocViewRec view of *xs*.

We can provide a covering function snocViewRec with the following type. Its implementation is shown in Figure 2.9.

```
snocViewRec : (xs : List a) → SnocViewRec xs
```

This allows us to write a more efficient implementation of reverse, using the notation introduced in Section 2.2.5.2.

```
reverse : List a → List a
reverse     xs          with snocViewRec xs
reverse     [Nil]       | SNil              = Nil                              (2.40)
reverse  [xs ++ x :: Nil] | SSnoc {xs} sxs x = x :: reverse xs | sxs
```

This implementation calculates the view only once, which takes linear time, and then traverses it once, using the symbol | to provide the appropriate subview explicitly in

each step.

Computing the recursive snoc view of a list is equivalent to reversing the list – except that the extra typing information keeps the links between the original list and its reversed form.

Advantages of recursive views Recursive views therefore give us both components needed to implement the programs we want.

- Views present data in a way that is useful for the programs that we want to write (e.g. halves instead of head and tail).
- The recursive nature of recursive views means that a view intrinsically contains a termination argument for a function that recurses over it.

Laziness In languages that are strict by default but feature optional laziness, such as Idris, it may be beneficial to make recursive occurrences of views in their constructors lazy. This will avoid computing the full view in advance; instead, the view will be constructed as the function progresses, while still making the function pass the termination check.

2.2.8.3 Example: V-views

Suppose that we need to check whether a list is a palindrome and if it is, produce a proof of the fact. The predicate "is a palindrome" is encoded by the following type family.

```
data Palindrome : List a → Type where
  PNil  : Palindrome Nil
  POne  : Palindrome (x :: Nil)
  PMore : Palindrome mid → Palindrome (x :: mid ++ x :: Nil)
```

The view SnocViewRec lets us access the end of the list. However, in this case, we need access to *both* ends at the same time. We can get that using a V-view.

```
data VView : List a → Type where
  VNil  : VView Nil
  VOne  : (x : a) → VView (x :: Nil)
  VMore : (l : a) → (midV : VView mid) → (r : a) → VView (l :: mid ++ r :: Nil)
```

The name "V-view" comes from the visualisation of a list folded in the middle to obtain a V-shaped structure, as shown in Figure 2.10.

Then, given a covering function, whose implementation I omit,

```
vView : (xs : List a) → VView xs,
```

(A) Linear list (B) V-view of a list

FIGURE 2.10: V-views

we can implement the palindrome (semi-)decision procedure as follows[6].

```
isPalin : (xs : List a) → Maybe (Palindrome xs)
isPalin xs with vView xs
isPalin [Nil]          | VNil          = Just PNil
isPalin [x :: Nil]     | VOne x        = Just POne
isPalin [l :: mid ++ r :: Nil] | VMore l midV r  with decEq_ℕ l r
isPalin [l :: mid ++ r :: Nil] | VMore l midV r  | No contra  = Nothing
isPalin [l :: mid ++ l :: Nil] | VMore l midV [l] | Yes [Refl] =
  case isPalin mid | midV of
    Nothing ⇒ Nothing
    Just pf ⇒ Just (PMore pf)
```

Again, this view removes the necessity to traverse the whole list to get both the last element *and* the first element in each recursive step.

2.2.8.4 Example: Binary numbers

Unary natural numbers, as usually defined in dependently typed languages, (Listing 2.1) are very inefficient – representing a number with magnitude 1000 with a 1000-element linked list is wasteful and will be too slow for any computation with larger numbers.

We can therefore define a binary view of unary numbers, called Bin.

```
data Bit : ℕ → Type where
  O : Bit 0
  I : Bit 1

data Bin : (width : ℕ) → (value : ℕ) → Type where
  N  : Bin Z Z
  (#) : Bin w n → Bit b → Bin (S w) (b + n + n)
```

[6]The careful reader will notice that *mid* is not bound anywhere in the function. Here, I use a feature of Idris that figures out that *mid* should be bound to a particular implicit argument of VMore. It deduces that from the occurrence of *mid* in the forced list.

```
data Sum : ℕ → ℕ → ℕ → Type where
  MkS : Bit hi → Bit lo
        → c + x + y ≡ lo + hi + hi
        → Sum c x y

adb : Bit c → Bit x → Bit y → Sum c x y
adb I I I = MkS I I Refl
adb I I O = MkS I O Refl
adb I O I = MkS I O Refl
adb I O O = MkS O I Refl
adb O I I = MkS I O Refl
adb O I O = MkS O I Refl
adb O O I = MkS O I Refl
adb O O O = MkS O O Refl

lemmaB : (c + b + b' ≡ lo + hi + hi)
  → lo + (hi + n + n') + (hi + n + n') ≡ c + (b + n + n) + (b' + n' + n')
— body omitted

adc : Bit c → Bin w m → Bin w n → Bin (S w) (c + m + n)
adc c N N = N # c
adc c (xs # x) (ys # y) =
  case adb c x y of
    ⌈MkS⌉ hi lo eq ⇒ subst (Bin _) (lemmaB eq) (adc hi xs ys # lo)

addBin : Bin w m → Bin w n → Bin (S w) (m + n)
addBin = adc O
```

FIGURE 2.11: Implementation of the binary adder

With this view, we can convert numbers back and forth and perform computation in the domain that is more suitable for the task at hand. For example, we can write a binary adder function whose type guarantees that it returns the correct result.

```
addBin : Bin w m → Bin w n → Bin (S w) (m + n)
```

The implementation of addBin can be found in Figure 2.11.

2.2.8.5 Example: Mergesort

Another class of list views are "divide and conquer" views corresponding to algorithms like the Fast Fourier Transform [Cap01], Mergesort, and (functional) Quicksort. Let us implement Mergesort.

The non-recursive view Split in Listing 2.34 has the same drawbacks as SnocView in Section 2.2.8.1 – it has to be recalculated in every recursive step (which is fine in the case of Mergesort because it does not affect its complexity) but most importantly – it is not possible for functions to recurse on the calculated sublists and still be obviously terminating.

We will therefore define a recursive variant of Split in the same way as we did with the snoc view.

```
data SplitRec : List a → Type where
  SRNil  : SplitRec Nil
  SROne  : (x : a) → SplitRec (x :: Nil)
  SRMore : (x : a) → (xs : List a) → (y : a) → (ys : List a)
           → (sxs : SplitRec (x :: xs)) → (sys : SplitRec (y :: ys))
           → SplitRec (x :: xs ++ y :: ys)
```
(2.41)

Again, the type of the view SplitRec does not prescribe that the list is split into (approximate) halves.

Although SplitRec can represent any recursive split of a list, the covering function halveRec below produces one particular split of a list where the list is halved every time. It uses the functions halve (Figure 2.5) and shorterL/shorterR (Figure 2.6).

```
halveRec : (xs : List a) → SplitRec xs
halveRec xs with wf⊏length xs
halveRec xs | acc with halve xs
halveRec [Nil] | _ | SNil = SRNil
halveRec [x :: Nil] | _ | SOne x = SROne x
halveRec [y :: ys ++ z :: zs] | ⌈MkAcc⌉ acc | SMore y ys z zs
  = SRMore y ys z zs
      (halveRec _ | acc _ (shorterL {xs = y :: ys}))
      (halveRec _ | acc _ (shorterR {x = z}{ys = z :: zs}))
```

Now we can implement msort as follows, using the function merge from Listing 2.35.

```
msort : List ℕ → List ℕ
msort xs with halveRec xs
msort [Nil] | SRNil = Nil
msort [x :: Nil] | SROne x = x :: Nil
msort [x :: xs ++ y :: ys] | SRMore x xs y ys sxs sys
  = merge (msort (x :: xs) | sxs) (msort (y :: ys) | sys)
```

Operational behaviour of views The operational behaviour of msort is now different from what the code might suggest at first sight. The function does take a list and returns a list – but internally, it works in a very different way. First, it turns the list into its SplitRec view (and never calls halveRec again). Then, it recurses on the *view*, keeping the list just as an index of the view, despite its prominent position in the definition.

The recursive call "msort $(x :: xs)$ | sxs" may look like a recursive sorting of $(x :: xs)$ – and it's functional behaviour is exactly that. However, operationally, it ignores the list itself and works on the view of the list, sxs.

The same holds for the function reverse in Listing 2.40.

2.2.8.6 Recursive views with choice

The constructor SRSnoc of the view SnocViewRec in Listing 2.39 contains one recursive argument. The constructor SRMore of the view SplitRec in Listing 2.41 contains two recursive arguments. In both cases, the recursive arguments appear with fixed indices, which means that the view exactly prescribes the form of the recursive call.

However, sometimes we would like to vary the form of the recursive call – for example, we may not know in advance *where* to split the list, which happens in situations like deserialisation. In such cases, we can observe that a pair of recursive arguments, such as those of SRMore, can be represented as a function $(tag : \mathsf{Bool}) \to T(tag)$, where tag is either True (left component) or False (right component) and $T(tag)$ is a tag-dependent type of the component[7].

This generalises to the notion of an n-tuple being a function $(i : I) \to T(i)$, where i comes from an index set of a certain (possibly infinite) size, and $T(i)$ gives the component type.

We can therefore encode the type of split views *with choice*, named SplitRecC.

```
data SplitC : (List a → Type) → List a → Type where
  SCNil  : SplitC rec Nil
  SCOne  : (x : a) → SplitC rec (x :: Nil)
  SCMore : (x : a) → (xs : List a)
           → (y : a) → (ys : List a)
           → (rxs : rec (x :: xs))
           → (rys : rec (y :: ys))
           → SplitC rec (x :: xs ++ y :: ys)

data SplitRecC : List a → Type where
  SRC : (splitAt : (n : ℕ) → SplitC SplitRecC xs) → SplitRecC xs
```

Now the constructor SRC contains a function that gives us a split at the chosen point. Since we will always want to split lists into smaller parts, in *splitAt* n xs, we define that $n = 0$ means "split xs into head and tail" and $n \geq (\mathsf{length}\ xs - 2)$ means "split xs into all-but-last and last element".

Building the view First, we define non-recursive splitting. For that, we can reuse the type family SplitC with parameter $rec = \lambda\_.\ \mathsf{Unit}$.

```
pushSC : (x : a) → SplitC (λ_. Unit) xs → SplitC (λ_. Unit) (x :: xs)
pushSC x  SCNil                 = SCOne x
pushSC x  (SCOne y)             = SCMore x Nil y Nil () ()
pushSC x  (SCMore y ys z zs () ()) = SCMore x (y :: ys) z zs () ()
```

[7] I use parentheses because $T(i)$ is a function in the metalanguage.

```
splitAt : (n : ℕ) → (xs : List a) → SplitC (λ_. Unit) xs
splitAt    n          Nil        = SCNil
splitAt    n       (x :: Nil)    = SCOne x
splitAt    Z    (x :: y :: ys)   = SCMore x Nil y ys () ()
splitAt  (S n)  (x :: y :: ys)   = pushSC x (splitAt n (y :: ys))
```

Having non-recursive splitting defined, we can arrange it recursively using well-founded recursion. This time we use SplitC with parameter *rec* = SplitRecC.

```
lemmaApp : (xs : List a) → (ys : List a) → length ys ≤ length (xs ++ ys)
lemmaApp    Nil     ys  = refl≤
lemmaApp  (x :: xs)  ys  = leS (lemmaApp xs ys)

splitC : (xs : List a) → (n : ℕ) → SplitC SplitRecC xs
splitC          xs          n  with wf⊏ xs
splitC          xs          n  | acc          with splitAt n xs
splitC         [Nil]        n  | acc          | SCNil                = SCNil
splitC      [x :: Nil]      n  | acc          | SCOne x              = SCOne x
splitC  [y :: ys ++ z :: zs] n  | MkAcc acc   | SCMore y ys z zs () ()
  = SCMore y ys z zs
      (SRC (λn'. splitC (y :: ys) n' | acc _ (LES shorterL)))
      (SRC (λn'. splitC (z :: zs) n' | acc _ (LES (lemmaApp ys (z :: zs)))))

splitRecC : (xs : List a) → SplitRecC xs
splitRecC xs = SRC (splitC xs)
```

The function splitRecC is therefore the covering function for our recursive view with choice, SplitRecC.

Using the view We can use the view for tasks like deserialisation, where the split point is not known in advance. Suppose we have the following serialisation function that serialises a list of natural numbers to one number containing the length of the list, followed by the list itself. The whole stream is terminated with a zero-length list, which is not part of the encoded information.

```
packL : List (List ℕ) → List ℕ
packL     Nil      = Z :: Nil
packL  (xs :: xss) = length xs :: xs ++ packL xss
```

Using the view SplitRecC, we can implement the deserialisation function unpackL as follows.

```
unpack' : ℕ → List ℕ → List (List ℕ)
unpack' x       xs       with splitRecC (x :: xs)
unpack' x       xs       | ⌈SRC⌉ splitAt  with splitAt x
unpack' x       [Nil]    | _              | SCOne x  = Nil — dangling length tag
unpack' x  [ys ++ z :: zs] | _              | SCMore [x] ys z zs rys rzs
    = ys :: (unpack' z zs | rzs)

unpackL : List ℕ → List (List ℕ)
unpackL    Nil     = Nil
unpackL  (x :: xs) = unpack' x xs
```

The function unpack′ takes a non-empty list $x :: xs$ and uses its first element, x, as the split position. Notice how the types ensure that the first argument of SCMore, which is a recursive split of $(x :: xs)$, is forced to be equal to x.

Now we can check that unpackL $[1,3,3,1,2,3,2,0,1,4,1,2,3,4,0]$ evaluates to $\big[[3],[1,2,3],[0,1],[1,2,3,4]\big]$.

2.2.9 Summary

Views are a convenient way of reshaping data to fit the access pattern and recursive structure of the desired computation.

The function that constructs a view of a value is called the "covering function" of the view.

In recursive views in strict-by-default languages, it may pay off to make the recursive occurrences lazy to compute the views as we go, rather than in advance.

Well-founded recursion and domain predicates In Sections 2.2.6 and 2.2.7, we saw two related ways of implementing non-structural recursion. In both cases, recursive calls were made with an extra value: the termination argument.

The recursive calls usually have the form $f\ x_1 \ldots x_n \mid acc$, where $f\ x_1 \ldots x_n$ is the recursive call as it would be written in a language like Haskell, and *acc* is "the reason why the recursive call is terminating". This particular piece of syntax sugar that Idris introduced, the | notation, makes the recursive call so neat.

If a type has a *size* mapping from its elements into the natural numbers, we can easily generate accessibility predicates using this mapping. A useful instance is length and lists.

It is also remarkable that in all cases of well-founded recursion and domain predicates, all components of the termination argument are either forced in a pattern match, or unused, or passed down to recursive calls as sub-termination arguments.

Brady, McBride, and McKinna observe that accessibility/domain predicates can always be erased, which "restores the original operational behaviour of the program by computation on the indices". [BMM04]

Therefore, since *acc* is erasable, a recursive call with the form $f\ x_1 \ldots x_n \mid acc$ compiles to just $f\ x_1 \ldots x_n$ – exactly the recursive call that we intended to write. Using the sugar of with clauses, we achieved something that looks like a compiler-supported termination argument but in fact desugars to ordinary core code.

My erasure approach in Chapter 5 can spot and erase all these termination arguments, while the approach in Chapter 4 can spot and erase some of them – those *without choice* (see below); in other words, those that do not involve higher-order functions. (I do not give a formal proof.)

Views Views are different from accessibility predicates in that they are *informative*. They are designed to be an alternative representation of data, not just a proof of termination, although they may include the latter. Therefore they contain all the data that was contained in the original structure.

In this case, a call of the form $f\ x_1 \ldots x_n \mid view$ is operationally a call on the *view*, while $x_1 \ldots x_n$ are just its indices (see Section 2.1.4.4), and as we will see later, they will generally be erased.

In Section 2.2.8.1, we saw an example of a non-recursive view, which made writing functions more convenient but did not help us with writing recursive functions.

Section 2.2.8.2 introduced recursive views, which were structured recursively in exactly the same way as our algorithms required. This makes termination trivial – all recursion is structural over the view.

Finally, Section 2.2.8.6 introduced recursive views with choice, which do not fix the form of the recursive call in advance and instead give a sub-view for any possible recursive call.

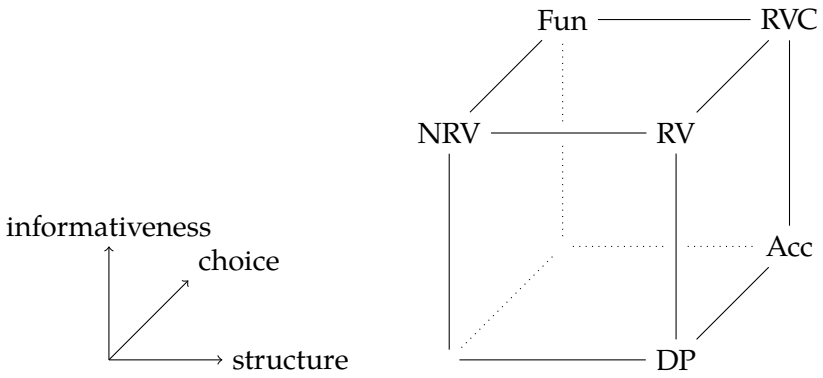
2.2.9.1 The view cube

Let us additionally extend the word "view" to include accessibility predicates and domain predicates. Then we can classify views by three criteria.

Structure Does the view have a recursive structure that we could possibly exploit as a proof of termination when writing recursive programs?

Choice If the view has a recursive structure, is it fixed in advance or can we choose how it branches locally at the point of a recursive call? Is it tailored to a particular function or is it meant to be more reusable?

For finite branching, there is theoretically no distinction between views with choice and views without it. In practice, a view that contains a function that computes sub-views is a view with choice.



Vertex	Description	Section
RVC	Recursive views with choice	2.2.8.6
RV	Recursive views	2.2.8.2
DP	Domain predicates	2.2.7
Acc	Accessibility predicates	2.2.6
NRV	Non-recursive views	2.2.8.1
Fun	Ordinary function calls	—

FIGURE 2.12: The view cube

Informativeness Does the view contain all the information present in the original data and do we regard it as its alternative representation?

Or does it contain no interesting information and can we erase it before running the program?

This criterion is not entirely intrinsic to the view and is mostly related to how it is (intended to be) used. An informative view can be used as non-informative by simply ignoring its informative parts (but not vice versa).

These three criteria classify views into a view cube (Figure 2.12). The corner "Fun" represents "views" that are informative, non-structural, and allow the user choose the form of the call. This description fits ordinary function calls.

Chapter 3

Motivation

3.1 Why we need erasure

Although it may not be obvious in their surface languages thanks to elaboration, dependently typed programs contain a lot of information, such as type annotations or values of implicit arguments.

However, not all information contained in programs is necessary to perform the computation that the programs describe. Often, a substantial part of a program actually consists of evidence that certain (undecidable) requirements of are satisfied, which allows efficient verification of type-correctness. This information is not necessary for runtime.

In general, it is undecidable to tell which parts of the program are "useful" and "computational", and which parts are "evidence" and "typechecking-only" – and sometimes there are even several different valid choices. The safest choice is therefore to compile and execute all programs as entirely computational, where their non-computational parts behave like normal code, taking up time and space. In other words, such programs run slower and take up more memory than necessary.

This becomes a significant problem when the non-computational parts start causing non-negligible performance overhead, or even become *asymptotically* more expensive than the "useful" parts of the program, as we will see later in the examples in Sections 3.1.2.2 and 3.1.2.3. In such cases, we need to try harder to identify the non-computational parts and erase them.

In this section, I elaborate on the above and give a few illustrative examples.

3.1.1 Type checking vs. execution

Once a program has been elaborated, we can perform all necessary checks, such as type checking, on its fully explicit form. However, a lot of the information that is necessary for (type-) checking is not necessary for evaluation, as shown in the following example.

$$\mathsf{id} : \{a : \mathsf{Type}\} \to (x : a) \to a$$
$$\mathsf{id}\ \{a\}\ x = x$$

The function id ignores its first argument a and always returns its second argument, x. Vector concatenation is similar.

$$
\begin{aligned}
&(+\!\!+) : \mathsf{Vect}\ m\ a \to \mathsf{Vect}\ n\ a \to \mathsf{Vect}\ (m+n)\ a \\
&(+\!\!+)\quad \mathsf{Nil}\quad ys\ =\ ys \\
&(+\!\!+)\ (x :: xs)\ ys\ =\ x :: (xs +\!\!+ ys)
\end{aligned}
$$

Function $(+\!\!+)$ elaborates to the following fully explicit form, where it is not clear that the length indices m and n are unnecessary for computation[1].

$$
\begin{aligned}
&(+\!\!+)\ :\ \{a : \mathsf{Type}\} \to \{m : \mathbb{N}\} \to \{n : \mathbb{N}\} \\
&\qquad \to \mathsf{Vect}\ m\ a \to \mathsf{Vect}\ n\ a \to \mathsf{Vect}\ (m+n)\ a \\
&(+\!\!+)\ \{a\}\ \{\mathsf{m} = [\mathsf{Z}]\}\quad \{n\}\ \mathsf{Nil}\ ys = ys \\
&(+\!\!+)\ \{a\}\ \{\mathsf{m} = \lceil \mathsf{S} \rceil\ k\}\ \{n\}\ \big((::)\ \{\mathsf{a} = [a]\}\ \{\mathsf{n} = [k]\}\ x\ xs\big)\ ys \\
&\quad = (::)\ \{a\}\ \{k+n\}\ x\ \big((+\!\!+)\ \{a\}\ \{k\}\ \{n\}\ xs\ ys\big)
\end{aligned}
$$

Morally, we ought to be able to calculate the concatenation of vectors just by looking at the vectors – we should not need their length indices for that.

Indeed, we could reimplement vector concatenation without arguments a, m and n, if we accept a weaker type signature with lists in a hypothetical language with static but imprecise types.

$$
\begin{aligned}
&(+\!\!+) : \mathsf{List} \to \mathsf{List} \to \mathsf{List} \\
&(+\!\!+)\quad \mathsf{Nil}\quad ys\ =\ ys \\
&(+\!\!+)\ (x :: xs)\ ys\ =\ x :: (xs +\!\!+ ys)
\end{aligned}
$$

While we got rid of the extra arguments, the type signature no longer guarantees anything about the type of the elements contained within the list, nor does it say anything about the lengths of the lists involved, although the implementation still works correctly.

However, it would be impossible to implement the vector concatenation function without one (or both) vector arguments. These are essential for computation.

Compile time vs. run time Another view to look at the problem is that we introduce extra arguments just to be able to express or link different parts of type signatures.

In the case of id, we did not introduce the type argument a in order to use it for computation in the function. We added it in order to create a name that mediates the equality of the type of x and the return type of the function in its type signature.

In the case of vector concatenation, we introduced the arguments a, m and n in order to be able to express that vector concatenation accepts two vectors that have the same element type and any length, and it returns a vector that also has the same element type and its length is the sum of the lengths of the inputs.

[1] They are unnecessary because their constructors are forced and all values projected out of them end up in unused positions in recursive calls.

In terms of stages of compilation, the arguments a, m and n are there "only for typechecking" at compile time and we could conceivably erase them from the program before executing it. On the other hand, the remaining two vector arguments are necessary for execution and must exist at run time.

3.1.1.1 Notation

To distinguish the two phases of program lifetime, I use the following notation.

R, runtime The letter R stands for computational values needed for execution. These have to be **r**etained for **r**untime.

E, erasable The letter E stands for non-computational values that can be erased from the program without affecting its execution.

I avoid the term "irrelevant", which I use exclusively for the principle whereby two irrelevant values are definitionally equal, as mentioned in Section 2.1.7.

3.1.2 Examples

In Section 1.2.1, I showed an example of a program that "ought to" run efficiently, along with experimental indication that it does not. Let us examine more examples of programs where precise type signatures incur performance overhead.

I will assume strict evaluation With strict evaluation, it is easier to reason about performance and demonstrate the points I want to make. However, the problem is still present with lazy evaluation (Section 3.2.2.3).

3.1.2.1 Vectors

Let us start with a function that computes the parity of a concatenation of two Boolean vectors.

The function named parity XORs the elements of any given vector and returns the result, assuming that $(\oplus) : \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}$ is the exclusive-or operation.

$$
\begin{aligned}
&\mathsf{parity} : \mathsf{Vect}\ n\ \mathsf{Bool} \to \mathsf{Bool} \\
&\mathsf{parity}\quad \mathsf{Nil}\quad = \mathsf{True} \\
&\mathsf{parity}\ \ (x :: xs)\ = x \oplus \mathsf{parity}\ xs
\end{aligned}
$$

We will also need the vector concatenation function.

$$(+\!\!+) : \mathsf{Vect}\ m\ a \to \mathsf{Vect}\ n\ a \to \mathsf{Vect}\ (m + n)\ a$$

Finally, let us have two vectors of some lengths.

$$
\begin{aligned}
m &: \mathbb{N} \\
n &: \mathbb{N} \\
xs &: \mathsf{Vect}\ m\ \mathsf{Bool} \\
ys &: \mathsf{Vect}\ n\ \mathsf{Bool}
\end{aligned}
$$

The question is: how much work do we need to perform in order to evaluate the following expression?

$$\mathsf{parity}\ (xs +\!\!+\ ys)$$

It might seem that all we need to do is to prepend xs to ys, which takes m steps, and then calculate the parity of the resulting vector by traversing all its $m + n$ elements.

However, elaboration reveals another bit of computation that we need to perform.

Elaboration This is the elaborated definition of parity.

$$
\begin{aligned}
&\mathsf{parity} : \{n : \mathbb{N}\} \to \mathsf{Vect}\ n\ \mathsf{Bool} \to \mathsf{Bool} \\
&\mathsf{parity}\ \ \{\mathsf{n} = [\mathsf{Z}]\}\ \ \big(\mathsf{Nil}\ \{\mathsf{a} = [\mathsf{Bool}]\}\big) &&= \ \mathsf{True} \\
&\mathsf{parity}\ \ \{\mathsf{n} = \lceil \mathsf{S} \rceil\ k\}\ \ \big((::)\ \{\mathsf{a} = [\mathsf{Bool}]\}\ \{\mathsf{n} = [k]\}\ x\ xs\big) &&= \ x \oplus \mathsf{parity}\ \{k\}\ xs
\end{aligned}
$$

We will also need the elaborated type of $+\!\!+$.

$$(+\!\!+) : \{m : \mathbb{N}\} \to \{n : \mathbb{N}\} \to \mathsf{Vect}\ m\ a \to \mathsf{Vect}\ n\ a \to \mathsf{Vect}\ (m + n)\ a$$

The expression that we want to evaluate elaborates to the following.

$$\mathsf{parity}\ \{m + n\}\ \big((+\!\!+)\ \{m\}\ \{n\}\ xs\ ys\big)$$

The above reveals that in order to invoke parity on $(xs +\!\!+\ ys)$, we need to calculate $m + n$ as well.

This overhead does not make the whole operation *asymptotically* worse but it is also not entirely negligible: concatenation of xs and ys takes $O(m)$ steps but addition of unary numbers $m + n$ *also* takes $O(m)$ steps; see the definition of $(+)$ in Listing 2.2.

3.1.2.2 Palindrome

In Section 2.2.8.3, we saw a function that checks whether a list is a palindrome. It does so by constructing a V-view of a list, represented by the following type family.

$$
\begin{aligned}
&\textbf{data}\ \mathsf{VView} : \mathsf{List}\ a \to \mathsf{Type}\ \textbf{where} \\
&\quad \mathsf{VNil}\ \ : \mathsf{VView}\ \mathsf{Nil} \\
&\quad \mathsf{VOne}\ \ : (x : a) \to \mathsf{VView}\ (x :: \mathsf{Nil}) \\
&\quad \mathsf{VMore} : (l : a) \to (midV : \mathsf{VView}\ mid) \to (r : a) \to \mathsf{VView}\ (l :: mid +\!\!+\ r :: \mathsf{Nil})
\end{aligned}
$$

It turns out that this view *cannot be constructed in linear time*. To understand why, let us make the argument *mid* of VMore explicit and look at how a V-view of the list 1 :: 2 :: 3 :: 4 :: 5 :: 6 :: 7 :: Nil is represented.

```
VMore {mid = 2 :: 3 :: 4 :: 5 :: 6 :: Nil} 1 (
  VMore {mid = 3 :: 4 :: 5 :: Nil} 2 (
    VMore {mid = 4 :: Nil} 3 (
      VOne 4
    ) 5
  ) 6
) 7
```

The indices *mid* (highlighted red) form a triangular structure, which means that they take up quadratic space and therefore the whole view takes up quadratic space in the length of the list. Since there is no way to produce a quadratically sized structure in linear time, the function that computes the view (its covering function) must take at least quadratic time to do so.

Sharing does not help Not all "triangular shapes" are necessarily quadratic in concrete representation, as witnessed by the function tails that returns a list of all suffixes of the given list (including the whole list).

```
tails : List a → List (List a)
tails    Nil    = Nil :: Nil
tails (x :: xs) = (x :: xs) :: tails xs
```

The function tails can exploit *sharing*, where all tails of a list can be efficiently represented in a linearly sized structure, as shown in Figure 3.1.

However, in the case of V-views, the lists in the index *mid* cannot share spines because they do not have a common suffix.

3.1.2.3 Binary adder

In Section 2.2.8.4, we saw a function that adds two binary numbers.

```
data Bit : ℕ → Type where
  O : Bit 0
  I : Bit 1

data Bin : (width : ℕ) → (value : ℕ) → Type where
  N   : Bin Z Z
  (#) : Bin w n → Bit b → Bin (S w) (b + n + n)

addBin : Bin w m → Bin w n → Bin (S w) (m + n)
```

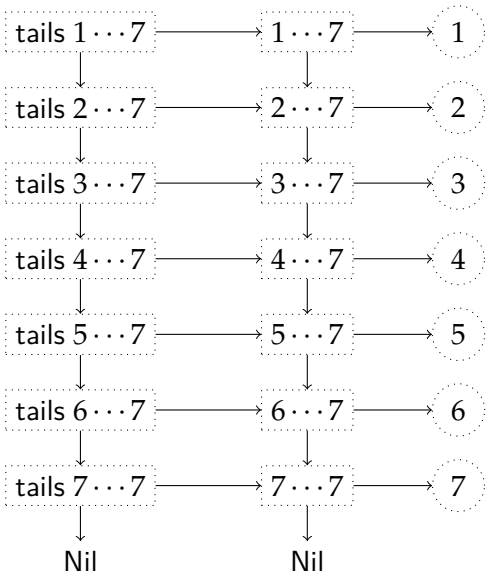


FIGURE 3.1: Shared tails of (1 :: 2 :: 3 :: 4 :: 5 :: 6 :: 7 :: Nil)
Each box represents a list (a cons cell), where the rightward arrow points to the head and the downward arrow points to the tail of the list.

Clearly, this function ought to run in $O(w)$ time, where w is the number of bits in the operands. However, it turns out that it runs in $O(2^w)$ time.

Like in the example with V-views, the problem is hidden in the implicit arguments to the constructors. Namely, the constructor (#) contains the index $\{n : \mathbb{N}\}$, which is the value of the binary number, represented in *unary*.

$$(\#) : \{w : \mathbb{N}\} \to \{n : \mathbb{N}\} \to \{c : \mathbb{N}\} \to \mathsf{Bin}\ w\ n \to \mathsf{Bit}\ b \to \mathsf{Bin}\ (\mathsf{S}\ w)\ (b + n + n)$$

Being a unary number, the index n occupies memory size that is *exponential* in the number of bits. Thus, just constructing a binary number with w bits takes $O(2^w)$ time.

Arithmetic on the indices It gets worse still. Like in Section 3.1.2.1, we can define a parity function for binary numbers.

$$\mathsf{parity} : \mathsf{Bin}\ w\ n \to \mathsf{Bool}$$

Then the expression that computes the parity of the sum of two numbers, $x : \mathsf{Bin}\ w\ m$ and $y : \mathsf{Bin}\ w\ n$, normally expressed as follows,

$$\mathsf{parity}\ (\mathsf{addBin}\ x\ y)$$

elaborates to the following.

$$\mathsf{parity}\ \{\mathsf{w} = w\}\ \{\mathsf{n} = m + n\}\ \big(\mathsf{addBin}\ \{w\}\ \{m\}\ \{n\}\ x\ y\big)$$

Therefore, whenever we pass the sum two binary numbers as an argument to another function, which should be an $O(w)$ operation, we also have to perform addition on

the two *unary* representations, which takes $O(2^w)$ time.

3.1.2.4 Run-length encoding

We can define a view of RLE-compressed lists, where long runs of identical elements can be represented by the length of the run and the value to repeat.

```
replicate : ℕ → a → List a
replicate  Z    x  =  Nil
replicate  (S n)  x  =  x :: replicate n x

data RLE : List a → Type where
  RNil  : RLE Nil
  RCons : (n : ℕ) → (x : a) → RLE xs → RLE (replicate n x ++ xs)
```

The function decompress is then defined as follows.

```
decompress : RLE xs → List a
decompress        RNil        =  Nil
decompress  (RCons n x rxs)  =  replicate n x ++ decompress rxs
```

The problem here is that we *have to pass the uncompressed list as an argument to decompress*. Indeed, the elaboration of (decompress *rxs*), where *rxs* : RLE *xs*, is the following term.

```
decompress {xs} rxs
```

It is actually straightforward to prove that decompress $\{xs\}$ *rxs* = *xs*.

Although the size of the RLE view is not necessarily asymptotically smaller than the size of the uncompressed list – especially when the run length is represented as a unary number – the fact that it always has to contain (keep references to) the uncompressed list is unintuitive and certainly undesirable.

3.1.3 Summary

I have shown a few examples of programs that look reasonable but have non-trivial runtime overhead caused by dependent typing: if we implemented them with less precise types, these performance problems would be eliminated.

- Vector concatenation incurs a non-constant overhead, which is however still within the bounds of asymptotic complexity of the concatenation operation itself.
- A V-view is not constructible in better than quadratic time (and space), while the expected complexity is linear. This means that we cannot use V-views to decide palindromicity in linear time.

Program	Complexity expected	Complexity actual	where n is
Binary adder	$O(n)$	$O(2^n)$	number of bits
Palindrome checker	$O(n)$	$O(n^2)$	length of list
Run-length decoder	$O(n)$	$O(n)$	length of input

TABLE 3.2: Expected and actual time/space complexities of the example programs

- Operations with binary numbers take exponential time and space in the number of binary digits, while the expected complexity is linear.
- The RLE decompression function requires the decompressed list as one of its arguments.

The above points are summarised in Table 3.2.

We need erasure If we want to use dependent types for realistic, practical functional programming, we certainly must remove this overhead. We ought not to pay this performance penalty for all the advantages of dependent types.

We therefore need a way to *remove* non-computational parts of programs before executing them, which in compiled languages means erasure before code generation. In other words, we need to identify the *phase distinction* in programs [Car88].

For reasonable usability, we also need an *automated way to recognise* which parts of a given program can be erased safely and which parts need to be preserved for run time. This is undecidable in general but this dissertation presents two (increasingly powerful) ways (Chapters 4 and 5) that have proven to work well in practice (Chapter 9).

3.2 Non-satisfactory approaches to erasure

There are several approaches to removing non-computational parts of programs that are used in current languages, and some approaches that one might come up with when thinking about the problem of erasure for a while. I will list some of them.

Indices vs. proofs In the following, I will refer to the distinction between *indices* and *proofs*.

An *index* is an argument of a type constructor used in a certain way, as described in Section 2.1.2.1. In all examples in Section 3.1.2, the problem was an *index* that was too big.

A *proof* is a value that represents evidence that a certain proposition holds. Its type (family), which represents the proposition, is usually itself indexed (or parameterised) by other values.

$$
\begin{array}{l}
\textbf{data}\ \mathsf{All} : (p : a \to \mathsf{Type}) \to \mathsf{List}\ a \to \mathsf{Type}\ \textbf{where} \\
\quad \mathsf{ANil}\ \ : \mathsf{All}\ p\ \mathsf{Nil} \\
\quad \mathsf{ACons}\ : p\ x \to \mathsf{All}\ p\ xs \to \mathsf{All}\ p\ (x :: xs)
\end{array}
$$

$$
\begin{array}{l}
\textbf{data}\ \mathsf{AllDifferent} : \mathsf{List}\ a \to \mathsf{Type}\ \textbf{where} \\
\quad \mathsf{ADNil}\ \ : \mathsf{AllDifferent}\ \mathsf{Nil} \\
\quad \mathsf{ADCons}\ : \mathsf{All}\ \big(\lambda y.\ \mathsf{Not}\ (x \equiv y)\big)\ xs \to \mathsf{AllDifferent}\ (x :: xs)
\end{array}
$$

$$
\begin{array}{lccccccccc}
xs_2 & \neq & & & & & & & & \\
xs_3 & \neq & \neq & & & & & & & \\
xs_4 & \neq & \neq & \neq & & & & & & \\
xs_5 & \neq & \neq & \neq & \neq & & & & & \\
xs_6 & \neq & \neq & \neq & \neq & \neq & & & & \\
xs_7 & \neq & \neq & \neq & \neq & \neq & \neq & & & \\
xs_8 & \neq & \neq & \neq & \neq & \neq & \neq & \neq & & \\
xs_9 & \neq & \neq & \neq & \neq & \neq & \neq & \neq & \neq & \\
xs_{10} & \neq & \neq & \neq & \neq & \neq & \neq & \neq & \neq & \neq \\
 & xs_1 & xs_2 & xs_3 & xs_4 & xs_5 & xs_6 & xs_7 & xs_8 & xs_9
\end{array}
$$

FIGURE 3.3: AllDifferent, the predicate expressing that all elements of a list are different, has a quadratic size

We could see an example of both in Section 2.1.4.4. The function $\mathsf{half} : (n : \mathbb{N}) \to (pf : \mathsf{Even}\ n) \to \mathbb{N}$ takes two arguments: the argument pf is a proof that n is even and n is its index.

While all examples in in Section 3.1.2 had indices that were too big, we can also have proofs that are too big. An example would be the function that inverts a mapping represented as a list of pairs, by flipping the pairs contained within.

$$\mathsf{invertMap} : \big(xs : \mathsf{List}\ (a, b)\big) \to \mathsf{AllDifferent}\ (\mathsf{map}\ \mathsf{snd}\ xs) \to \mathsf{List}\ (b, a)$$

The prerequisite for invertibility of a mapping is its injectivity. This is represented as a proof that all values of the mapping are distinct.

However, while we would expect invertMap to run in linear time, in order to invoke it, we need to provide a proof of injectivity, which will likely be quadratic in size, as shown in its usual implementation in Figure 3.3.

I do not talk about them in my dissertation because *proofs* (unlike indices) are mostly satisfactorily erased by the established systems.

3.2.1 Current systems

3.2.1.1 Idris, forcing and collapsing

The combination of the forcing, detagging and collapsing optimisations [BMM04] can erase some important classes of data, such as accessibility and domain predicates predicates (Sections 2.2.6 and 2.2.7).

However, it does not erase the indices in the palindrome example (Section 3.1.2.2), and neither does it help with the binary numbers example (Section 3.1.2.3).

Forcing Brady, McBride and McKinna define that a constructor argument is *forceable* if it is uniquely and entirely determined by the indices of its type family.

A constructor argument is *concretely forceable* if it can be reconstructed in constant time by pattern matching on the indices.

Concretely forceable constructor arguments can thus be erased because they are always available from elsewhere in a pattern match.

Detagging A type family is detaggable if the constructor tag is uniquely determined by the indices of the type family.

A type family is *concretely detaggable* if the indices of its constructors are disjoint in a way that is efficiently decidable.

Constructor tags can therefore be erased in concretely detaggable type families.

Replacing matching on constructor tags with matching on the indices may change the operational behaviour of the program but the authors do not elaborate further on when such optimisation is likely to be an improvement and when it is not.

Both methods that I introduce in this dissertation, in Chapter 4 and Chapter 5, let the user (or the elaboration algorithm) decide which operational semantics they want.

Collapsing A type family is called *collapsible* if any value belonging to that family is entirely determined by its indices.

If all non-recursive arguments of all constructors of a concretely detaggable type family are concretely forceable, we are left with only recursive arguments belonging to the same type family and we call such a family *concretely collapsible*.

We can observe that a concretely collapsible family is indeed collapsible because all recursive arguments are determined from *their* indices by induction, and their indices are themselves determined – given in terms of the other (determined) arguments.

A value belonging to a concretely collapsible type family therefore does not contain any information that is not easily recoverable from its indices and can therefore be erased.

Savings at compile time The advantage of the above optimisations is that they are applicable at *compile time* because they are "lossless"; they erase only duplicate information, not all unused information.

The paper [BMM04] also discusses slightly more aggressive erasure for runtime.

Shortcomings The forcing/detagging/collapsing optimisation is not useful for erasing the problematic data in the examples in Section 3.1.2.2 and Section 3.1.2.3 because the duplication there is not "concrete". Let us look at the definition of

V-views.

```
data VView : List a → Type where
  VNil  : VView Nil
  VOne  : (x : a) → VView (x :: Nil)
  VMore : (l : a) → (midV : VView mid) → (r : a) → VView (l :: mid ++ r :: Nil)
```

While we know that there is only one way to decompose the list $(l :: mid +\!\!+ r :: \text{Nil})$ into its three components, the computer cannot see that. Furthermore, the extraction of *mid* would certainly not be a constant-time operation, as required by concrete forceability.

One could conceive a forceability scheme that drops the requirement of constant-time index recomputation and allows the user to provide their own index recomputation functions in cases where they are not obvious. In addition, one could forbid accessing the constructor arguments that are not reconstructible, for selected type families.

Finally, the forcing/detagging/collapsing optimisation will not remove data that is not duplicated. However, we often want to erase "irreversibly" because we know we will not need *any* copy of the value at run time at all.

3.2.1.2 Coq and Prop

Coq [The04] features Prop, a universe of values designated for erasure during program extraction [Pau89; Let03; Let08]. The programmer decides whether a type belongs to Prop, the universe of erased values or to Type, the universe of unerased values.

Proofs The Prop universe works well for erasure of proofs, whose types will have to be in Prop. An example would be the predicate AllDifferent with the same meaning as we saw above – except that the target type of the type constructors All and AllDifferent is Prop.

```
Inductive All {a : Type} (p : a → Prop) : list a → Prop :=
  | ANil : All p nil
  | ACons : forall x xs, p x → All p xs → All p (cons x xs).

Inductive AllDifferent {a : Type} : list a → Prop :=
  | ADNil : AllDifferent nil
  | ADCons : forall x xs, All (fun y ⇒ x <> y) xs
    → AllDifferent xs
    → AllDifferent (cons x xs).
```

During program extraction, Coq will erase all values whose types are in Prop, which includes all proofs of AllDifferent. This can be done because the type checker does not allow elimination of values (whose types are) in Prop whenever (the type of) the

constructed value is in Type, except for empty and singleton elimination [The04]. This statically guarantees that there is no flow of information from Prop to Type.

Indices However, Prop is not useful for erasure of indices, which usually have a non-Prop type. As an example, we can give the type of binary numbers, as shown in Section 3.1.2.3. What should the type of *value* be in the following type signature?

$$\mathsf{Bin} : (\mathit{width} : \mathbb{N}) \to (\mathit{value} : \mathbb{N}) \to \mathsf{Type}$$

We have established in Section 3.1.2.3 that *value* must be erased in any reasonable program. But $\mathbb{N}$, in Coq known as nat, is in Type, not in Prop. We *could* define pnat, a "mirror" copy of nat in Prop.

```
Inductive pnat : Prop
  | PO : pnat
  | PS : pnat → pnat
```

Then we will have to redefine Bin to have the following type signature.

$$\mathsf{Bin} : (\mathit{width} : \mathbb{N}) \to (\mathit{value} : \mathsf{pnat}) \to \mathsf{Type}$$

However, this solution has its own problems because now the following function type signature does not typecheck.

$$\mathsf{natToBin} : (n : \mathbb{N}) \to \mathsf{Bin}\ w\ n$$

The type of n is $\mathbb{N}$ but Bin expects pnat. We however cannot change the type of n to pnat because clearly execution of natToBin depends on it. We could add a conversion function as follows,

$$\mathsf{natToBin} : (n : \mathbb{N}) \to \mathsf{Bin}\ w\ (\mathsf{nat2pnat}\ n)$$

but then the conversion function gets in the way of equality; we still have duplication of types and all corresponding functions: e.g. addition must be defined separately for nat and pnat; and this approach is incompatible with proof irrelevance.

Summary The key problem here is the *inflexible erasure semantics*, where erasability is an *intrinsic* property of a type [ML08], and we thus cannot use a single value with a single type in both erased and non-erased contexts.

Even though with effort, we can avoid duplication and achieve good erasure [Lei14b] with an approach in the spirit of the nc monad [LS05], it requires reformulation of the code, heavy automation, and its theoretical soundness is questionable [Lei14a; Doc14].

Finally, this approach makes erasure fully explicit and there is no erasure inference.

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

fishy : Bin 0   — value seems to represent 0
fishy = I 0 N   — in fact, value represents 1
```

(A) A type-correct Agda program using an irrelevant index of Bin.

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

id✗ : .(n : ℕ) → Bin n → Bin n
id✗ n x = x
```

(B) Bin with a relevant index and $\mathsf{id}_{\not x}$ with an irrelevant n does not typecheck.

FIGURE 3.4: Irrelevance vs. indices of type families in Agda

3.2.1.3 Agda and irrelevance

In Agda, types are not intrinsically erased or otherwise, but any binder can be made *irrelevant* (and thus erased) by putting a dot in front of it [Agd14]. In Agda, the function invertMap might have the following type signature.

$$\mathsf{invertMap} : \big(xs : \mathsf{List}\ (a, b)\big) \to .\big(\mathsf{AllDifferent}\ (\mathsf{map}\ \mathsf{snd}\ xs)\big) \to \mathsf{List}\ (b, a)$$

Irrelevance is intended for proofs, where we are interested only in inhabitance of types but not the exact proof terms. All members of an irrelevant type are automatically considered equal, pattern matching on them is not allowed, they can be used only in irrelevant contexts – and they are erased before code generation.

Indices Again, while irrelevance works for erasing *proofs*, it is not useful for erasure of indices, as illustrated in the type-correct program shown in Figure 3.4a. The problem is that two values of an irrelevant type are considered equal already at the point of typechecking, which defeats the point of indexing data types. The notion of irrelevance is therefore too strong (Section 2.1.7).

After the example of Barras and Bernardo's ICC* [BB08], Mishra-Linger's EPTS [ML08], and Zombie [Sjö15], we might also conceive the program in Figure 3.4b, where the index in the type declaration is relevant (because it does matter for the *type*) but an irrelevant n is passed into it in the function $\mathsf{id}_{\not x}$ (because it does not matter for the functional behaviour). This variation is considered in Section 2.1.7.2 but such programs are not accepted by the Agda type checker because their consistency with η-equality and large elimination is questionable [Abe11; Abe17]; see also Section 2.1.7.

Recent developments in Agda [AVW17] add another experimental form of irrelevance that allows formulation of the above, but irrelevance itself is still too strong a notion (Section 2.1.7) and irrelevance annotations are still fully explicit in Agda programs.

3.2.1.4 Zombie and irrelevance

Zombie [Sjö15] also uses irrelevance to achieve erasure. Since Zombie's equality is untyped, irrelevant values are more mobile than in Agda, and Zombie can erase all example programs shown in Section 3.1.2.

However, similar to Agda, Zombie equates erasure and irrelevance, which are distinct concepts (Section 2.1.7) and has no inference for them.

Zombie is discussed further in Section 8.2.3.

3.2.2 Other ideas

3.2.2.1 Erase exactly the types

Languages like Cayenne [Aug99] and Haskell [Jon03] erase exactly the types; Cayenne erases all values belonging to any type $t : \#_i$ where $i > 1$, while Haskell has a syntactic distinction between terms and types.

This is insufficient because all problematic values that we needed to erase in the examples in the above Section 3.1.2 are non-types.

Furthermore, that would preclude *typecase*, which we may want to have.

3.2.2.2 Erase all implicits

This strategy erases exactly the implicit arguments and make it a compile-time error to try to access them.

As a specific example where this is inconvenient is the text concatenation operator (coming from an Idris library).

$$(+\!\!+) : \{e : \mathsf{Encoding}\} \to \mathsf{Text}\ e \to \mathsf{Text}\ e \to \mathsf{Text}\ e$$

Here, the encoding is needed at run-time but we certainly want to keep it implicit because $+\!\!+$ is an infix binary operator.

Section 2.1.4.2 shows that we *could* actually erase e and have append project it out of the constructor of Text. However, this will not work if e is bound implicitly in the constructor, too. Furthermore, this limits the ways the program could work, and, for example, is in a direct contradiction with the forcing optimisation (Section 3.2.1.1).

The Hindley-Milner coincidence More generally, as Conor McBride argues [McB12; McB15], HM-style type systems traditionally conflate the distinctions between types vs. values, visible vs. invisible, programmer-specified vs. inferred, runtime vs. erased, non-dependent vs. dependent – McBride calls this "the Hindley-Milner coincidence". In a language like Haskell, *type* arguments to (polymorphic) functions are *invisible* (implicit), *inferred* by the typechecker, dependent, and *erased* before running the program. On the other hand, *non-type* arguments to functions are always *visible* (explicit), *programmer-provided*, non-dependent and *unerased*.

- terms vs types
- explicitly written things vs implicitly written things
- presence at run-time vs erasure before run-time
- non-dependent abstraction vs. dependent quantification
- parametric vs. non-parametric

However, in dependently typed languages, we often need to break this alignment and erase non-types (as shown earlier), retain implicit arguments for runtime (as shown here), give types explicitly (e.g. in $\mathsf{show} \circ \mathsf{read}$), etc. Erasing exactly the implicits – and thus tying explicitness to erasedness – would severely restrict the expressivity of the (surface) language.

For comparison, Dependent Haskell [Eis16] has *twelve* different quantifiers that express various combinations of the above options.

Furthermore, this would make erasure fully explicit in the input program, programmers would have to explicitly say what's erased all the time, and it would also be incompatible with erasure polymorphism.

Mishra-Linger [ML08, Sec. 7.3.3] also finds examples in literature where types are computational, proofs are computational, and non-types, non-proofs are non-computational.

3.2.2.3 Laziness

Laziness would stop the overhead from indices from getting asymptotically bigger but would not remove it completely. Furthermore, laziness is not applicable in strict languages.

Asymptotic overhead Since producing thunks is normally a constant-time operation even if the thunks describe expensive computation, laziness would prevent asymptotic slowdown from unnecessary index recomputation.

For example, in the example with binary numbers (Section 3.1.2.3), if the unary representation stored in the constructors were lazy (and never forced), we would avoid the exponential slowdown.

However, this assumes that the thunks are never forced, which brings new problems.

Non-asymptotic overhead Even if there is no asymptotic worsening, there is still overhead. The program has to have the relevant data stored *somewhere* and a long chain of unevaluated thunks is not more efficient than a long list.

Leaks memory If a long-running recursive function has a lazy argument that is never forced but updated in every iteration, it will accumulate thunks in it, leading to a memory leak.

Optimising compilers, such as GHC, can spot many cases of an unused argument and remove it. But this optimisation has nothing to do with laziness and would remedy the problem that we are trying to solve using laziness, without using laziness.

Forced patterns In dependently typed languages, values can be case-inspected but still unused. In Section 3.1.2.1, the elaborated form of parity shows that even if the constructor tag of its argument n is forced every time, we project k out of it in the second clause to use it in the recursive call, where it is unused.

This shows that forced patterns in dependently typed languages should be implemented using irrefutable patterns, as found in Haskell, to avoid unnecessary evaluation.

Not applicable in strict languages Finally, laziness is not an option in strict languages, where we cannot make all values lazy. However, deciding *which* values should be made lazy amounts to deciding which values are unused. However, if such analysis is available, it would be better to remove these values entirely.

3.2.2.4 Program reformulation

It may be possible to also completely reformulate the program and change its data structures and restructure its functions so that we do not need a more advanced erasure mechanism than is available currently.

Leivent shows [Lei14b] that one can achieve good erasure of indices in Coq using an approach similar to Letouzey's and Spitters's nc monad [LS05] and heavy automation, as mentioned in Section 3.2.1.2.

However, if the purpose of dependent types is to reduce the cost of writing correct software, having to turn one's program inside out, disabling the use of idioms like indexed views, and imposing other limitations defeats the purpose – especially when there is a better way, as shown in this dissertation.

3.3 Summary

This chapter has outlined why we need erasure and also what properties a desirable solution should have.

Flexible erasure semantics as defined by Mishra-Linger [ML08]. Erasability should not be an intrinsic property of a value (determined by its type, for example), but a contextual property depending on where the value occurs.

Operationally inspired erasure The problem we are trying to solve is inefficiency of programs at run time. While improvements to the performance of type checking

would be valuable, in this dissertation, I focus on run time performance of compiled programs.

Erasure inference/elaboration While the programmer should be able to explicitly request (non-)erasure of particular entities, the machine should be able to infer the rest.

Views and other dependently typed idioms The erasure scheme should work well with idioms like views or accessibility/domain predicates.

Practicality The solution should also be implementable in a practical language, ideally being based on an existing programming language, and should support or be compatible with features used in practical programming.

The following chapters describe two different approaches.

Chapter 4 shows a less powerful but simple erasure approach that can nevertheless address all problems in Section 3.1.2. This approach is currently implemented in Idris.

Chapter 5 shows a more systematic and more powerful approach to erasure, and its consequences are discussed in the rest of the dissertation.

Chapter 4

Untyped erasure

This chapter describes an erasure method that is simple and effective, although not without limitations – most importantly incomplete erasure from higher-order functions. The main advantage is that it is unintrusive and fits on top of an existing core calculus or intermediate representation, which made it easy to add it to Idris. Therefore, this is the erasure method that Idris 1 is using nowadays.

This analysis is effectively an untyped, flow-based *useless variable analysis* [Shi91], extended and tailored for the scenario of dependently typed programming languages. Most importantly, it supports inductive data types, erasing fields of data constructors independently, and pattern matching via case trees.

Unlike previous approaches to the problem of erasure (Section 3.2), this analysis can erase not only proofs but also indices, and supports the full Idris language including type classes, data structures and pattern matching. It is entirely automatic; no user annotations are required and no expressive power is lost.

Despite not being able to erase from higher-order functions (e.g. functions stored in data constructors), an important special case, type classes, is supported via a specialised extension (Section 4.5.3), and I sketch other extensions that would improve erasure of higher-order constructions (Section 4.5).

In this chapter, I first describe a core language consisting of bindings and case trees on which the analysis can be applied (Section 4.2). Despite being an intermediate representation of Idris, it is not specific to Idris and can therefore be used to implement erasure analysis for other languages. Furthermore, the analysis does not depend on the exact form or typing rules of the core language.

Then I present a *whole program* analysis (Section 4.3) which identifies the parts of functions and data structures which are *erasable* without affecting the result produced by the main function. This analysis need not be whole program if we restrict erasure inference for data constructors, as discussed in Section 4.7.1.

I propose several extensions to the presented erasure analysis method (Section 4.5), most of which are already implemented in Idris 1.

Finally, I demonstrate the effectiveness of the erasure analysis with several examples (Section 4.6) 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. I

then show that the experimental results match the theoretical expectations about asymptotical time complexity (Section 4.6.2).

4.1 Overview

This erasure approach relies on the fact that we can statically learn that some arguments of functions or data constructors do not affect the eventual value of main, as illustrated by the following examples.

4.1.1 Examples of unused values

Below, I show how erasure inference works on a few simple examples. For each example, I informally describe the process of extracting implications and show the final erased program.

The full process is described formally in Section 4.3.

4.1.1.1 Unreferenced variables

For example, it is easy to see that in the following function, the argument y is unused because it is never mentioned on the RHS.

$$\mathsf{const}^A : a \to b \to a$$
$$\mathsf{const}^A\ x\ y = x$$

How it works We can capture these relationships by the set of implications $\left\{\mathsf{const}_1^A \longleftarrow \{\mathsf{const}_\star^A\}\right\}$, which expresses that if const^A is used at all (represented by $\mathsf{const}_\star^A$), then its first argument (represented by const_1^A) is used.

Assuming that const^A is indeed used, which is represented by the implication $\mathsf{const}_\star^A \longleftarrow \{\}$, we obtain the set of implications $\left\{\mathsf{const}_1^A \longleftarrow \{\mathsf{const}_\star^A\}, \mathsf{const}_\star^A \longleftarrow \{\}\right\}$. If we regard this set of rules as a logic program and apply forward chaining to it, we obtain $\{\mathsf{const}_\star^A, \mathsf{const}_1^A\}$ as its minimal solution. The answer implies that const^A is used itself, and its first argument is used, too.

The erased program contains only the used portions:

$$\mathsf{const}^A\ x = x$$

The set $\{\mathsf{const}_\star^A, \mathsf{const}_1^A, \mathsf{const}_2^A\}$ is also a solution to the above set of implications, although it is not minimal. It corresponds to a variant of const^A, where the second argument is not erased – which is not optimal. On the other hand, $\{\mathsf{const}_\star^A\}$ is not a solution to this set of implication and it is also not a consistent erasure pattern.

This treatment may seem overly formal for const^A but we'll see that this approach gives answers in more complicated cases, too.

4.1.1.2 Unused arguments of functions

Here, we pass x into an unerased argument and y into an erased argument of const^A, and thus x and y can be erased accordingly.

$$
\begin{aligned}
&\mathsf{const}^B : a \to b \to a \\
&\mathsf{const}^B\ x\ y = \mathsf{const}^A\ x\ y
\end{aligned}
$$

How it works The implication set will contain $\mathsf{const}^A_\star \longleftarrow \{\mathsf{const}^B_\star\}$ since the invoked function is used whenever the invoking function is. For the arguments, we obtain $\mathsf{const}^B_1 \longleftarrow \{\mathsf{const}^B_\star, \mathsf{const}^A_1\}$ and $\mathsf{const}^B_2 \longleftarrow \{\mathsf{const}^B_\star, \mathsf{const}^A_2\}$. Argument x is used only if the whole RHS is used ($\mathsf{const}^B_\star$) *and* if const^A uses its first argument (const^A_1). If any of the two conditions is false, the application stops being a reason to mark x as used. The same holds for argument y.

Finally, we merge in the implications for const^A, obtaining the following set of implications.

$$
\begin{array}{lcll}
\mathsf{const}^A_\star & \longleftarrow & \{\mathsf{const}^B_\star\} & \\
\mathsf{const}^B_1 & \longleftarrow & \{\mathsf{const}^B_\star, \mathsf{const}^A_1\} & \\
\mathsf{const}^B_2 & \longleftarrow & \{\mathsf{const}^B_\star, \mathsf{const}^A_2\} & \\
\mathsf{const}^A_1 & \longleftarrow & \{\mathsf{const}^A_\star\} & \\
\mathsf{const}^B_\star & \longleftarrow & \{\} & \text{(assume } \mathsf{const}^B \text{ is used)}
\end{array}
$$

The resulting minimal solution is $\{\mathsf{const}^B_\star, \mathsf{const}^A_\star, \mathsf{const}^B_1, \mathsf{const}^A_1\}$, as expected. This solution does not contain const^A_2, nor does it contain const^B_2.

The erased form of const^B is therefore the following.

$$
\mathsf{const}^B x = \mathsf{const}^A x
$$

4.1.1.3 Loops in data flow

There are more interesting patterns we would like to recognise. In the following function, y is unused again, despite occurring on the RHS.

$$
\begin{aligned}
&\mathsf{const}^C : (n : \mathbb{N}) \to a \to b \to a \\
&\mathsf{const}^C\ \ \mathsf{Z}\ \ x\ y = x \\
&\mathsf{const}^C\ (\mathsf{S}\ n')\ x\ y = \mathsf{const}^C\ n'\ x\ y
\end{aligned}
$$

We can see that in the first clause, y does not occur on the RHS, and therefore if $n = \mathsf{Z}$, then the second argument of const^C is unused. In the second clause, y occurs only as the second argument of const^C, but by induction on n, this is an unused argument. The argument y is therefore always unused.

How it works In order to gather implications, we need to look at the elaboration of const^C into case trees. Let us disregard the implicit arguments for now.

$$
\begin{aligned}
&\mathsf{const}^C : \mathbb{N} \to a \to b \to a \\
&\mathsf{const}^C = \lambda n : \mathbb{N}.\ \lambda x : a.\ \lambda y : a. \\
&\quad \textbf{case}\ n\ \textbf{of} \\
&\qquad \mathsf{Z} \ \Rightarrow\ x \\
&\qquad \mathsf{S}\ n' \ \Rightarrow\ \mathsf{const}^C\ n'\ x\ y
\end{aligned}
$$

We can see that n is inspected immediately, which yields the implication $\mathsf{const}_1^C \longleftarrow \mathsf{const}_\star^C$. In the first branch, x is used, which is described by $\mathsf{const}_2^C \longleftarrow \mathsf{const}_\star^C$.

In the second branch, we however introduce a new variable n'. This variable is projected out of n, and thus its usage should imply const_1^C. However, its usage should also imply the usage of the first field of constructor S, represented by S_1. Each variable therefore may cause usage of multiple atoms, as a general principle, and we need to keep track of this mapping. Let us do it informally for now.

On the right hand side of the second branch, the function const^C is used unconditionally, which corresponds to the (not very exciting) constraint $\mathsf{const}_\star^C \longleftarrow \{\mathsf{const}_\star^C\}$. However, n' with its implied atoms $\{\mathsf{const}_1^C, \mathsf{S}_1\}$ is used only if const^C is used at all *and* const^C uses its first argument, which corresponds to the following implications, where we use sets of atoms to represent their conjunction.

$$
\begin{aligned}
\mathsf{const}_1^C &\longleftarrow \{\mathsf{const}_\star^C, \mathsf{const}_1^C\} \\
\mathsf{S}_1 &\longleftarrow \{\mathsf{const}_\star^C, \mathsf{const}_1^C\}
\end{aligned}
$$

From the remaining two arguments, we obtain implications $\mathsf{const}_2^C \longleftarrow \{\mathsf{const}_\star^C, \mathsf{const}_2^C\}$ and $\mathsf{const}_3^C \longleftarrow \{\mathsf{const}_\star^C, \mathsf{const}_3^C\}$.

If const^C is used at all, we obtain the following set of implications:

$$
\begin{aligned}
\mathsf{const}_\star^C &\longleftarrow \{\} \\
\mathsf{const}_1^C &\longleftarrow \{\mathsf{const}_\star^C\} \\
\mathsf{const}_2^C &\longleftarrow \{\mathsf{const}_\star^C\} \\
\mathsf{const}_\star^C &\longleftarrow \{\mathsf{const}_\star^C\} \\
\mathsf{const}_1^C &\longleftarrow \{\mathsf{const}_\star^C, \mathsf{const}_1^C\} \\
\mathsf{S}_1 &\longleftarrow \{\mathsf{const}_\star^C, \mathsf{const}_1^C\} \\
\mathsf{const}_2^C &\longleftarrow \{\mathsf{const}_\star^C, \mathsf{const}_2^C\} \\
\mathsf{const}_3^C &\longleftarrow \{\mathsf{const}_\star^C, \mathsf{const}_3^C\}
\end{aligned}
$$

Forward chaining shows that the minimal solution of this set of implications is $\{\mathsf{const}_\star^C, \mathsf{const}_1^C, \mathsf{const}_2^C\}$, which matches our informal argument above, and the erased form of const^C is therefore as follows.

$$
\begin{aligned}
&\mathsf{const}^C \quad \mathsf{Z} \quad x = x \\
&\mathsf{const}^C \ (\mathsf{S}\ n')\ x = \mathsf{const}^C\ n'\ x
\end{aligned}
$$

Erasure of constructor fields This example also illustrates why, seemingly redundantly, we qualify all implications with $\mathsf{const}^{C}_{\star}$. One might ask: clearly, if this function is used at all, the solution will be the same whether or not we include $\mathsf{const}^{C}_{\star}$ among the guards; if the function is not used, then it does not matter whether its arguments are erased or not.

However, the function may match on constructors and read their fields; consider the clause $\mathsf{pred}\ (\mathsf{S}\ n) = n$. Here, we want to mark the argument of S as used only if pred is used at all.

4.1.1.4 Pattern matching

So far, the erased argument was simply bound as a pattern variable on the LHS. However, in the following function, the argument n can still be erasable.

$$
\begin{aligned}
&\mathsf{vsum} : (n : \mathbb{N}) \to \mathsf{Vect}\ n\ \mathbb{N} \to \mathbb{N}\\
&\mathsf{vsum}\quad \mathsf{Z}\quad\ \ \mathsf{Nil}\quad\ = \mathsf{Z}\\
&\mathsf{vsum}\ \ (\mathsf{S}\ n')\ \ (x :: xs)\ = x + \mathsf{vsum}\ n'\ xs
\end{aligned}
$$

To see why, let us have a look at one possible elaboration of the pattern clauses into case trees.

$$
\begin{aligned}
&\mathsf{vsum} : (n : \mathbb{N}) \to \mathsf{Vect}\ n\ \mathbb{N} \to \mathbb{N}\\
&\mathsf{vsum} = \lambda n : \mathbb{N}.\ \lambda xs : \mathsf{Vect}\ n\ \mathbb{N}.\\
&\quad \textbf{case}\ xs\ \textbf{of}\\
&\qquad \mathsf{Nil}\quad\ \Rightarrow\ \textbf{case}\ n\ \textbf{of}\\
&\qquad\qquad\qquad\quad \mathsf{Z} \Rightarrow \mathsf{Z}\\
&\qquad x :: xs\ \Rightarrow\ \textbf{case}\ n\ \textbf{of}\\
&\qquad\qquad\qquad\quad \mathsf{S}\ n' \Rightarrow x + \mathsf{vsum}\ n'\ xs
\end{aligned}
$$

Both case inspections of n above have only one branch, and thus we do not need to inspect n to find out which branch to take. Furthermore, all variables projected out from n (namely n') end up in unused positions. Therefore we don't need any information from n at all, and it is unused as a whole.

Note that other elaborations into case trees would be possible. An elaboration that inspects n first would not be able to erase n. Because the value x projected out of the vector is always used, we would not be able to erase the argument xs, either. This is related to the choice described in Section 2.1.4.2.

How it works I will not list the implications coming from the function here, except for sketching the mechanism, which will be described more formally in the subsequent sections below.

While multi-branch case expressions immediately cause inspection of their scrutinee and produce an implication like $\mathsf{vsum}_1 \longleftarrow \{\mathsf{vsum}_\star\}$, single-branch case expressions do not. Instead, we just make a side note that usage of n' – if it is ever

used – should cause usage of vsum_1. Further usage analysis discovers that n' is never used, and thus neither is vsum_1. These useless case splits on n then *must* be removed before code generation.

The erased elaborated form of the above function is as follows.

$$
\begin{array}{l}
\mathsf{vsum} = \lambda xs. \\
\quad \textbf{case } xs \textbf{ of} \\
\qquad \mathsf{Nil} \quad \Rightarrow \mathsf{Z} \\
\qquad x :: xs \;\Rightarrow\; x + \mathsf{vsum}\ xs
\end{array}
$$

4.1.1.5 Fields of data constructors

If we stop ignoring implicit arguments, the full type of the vector constructor (::) is the following.

$$(::) : \{a : \mathsf{Type}\} \to \{n : \mathbb{N}\} \to (x : a) \to (xs : \mathsf{Vect}\ n\ a) \to \mathsf{Vect}\ (\mathsf{S}\ n)\ a$$

In the following function, which arguments are erasable?

$$
\begin{array}{l}
\mathsf{cons} : \{a : \mathsf{Type}\} \to \{n : \mathbb{N}\} \to (x : a) \to (xs : \mathsf{Vect}\ n\ a) \to \mathsf{Vect}\ (\mathsf{S}\ n)\ a \\
\mathsf{cons}\ a\ n\ x\ xs = (::)\ a\ n\ x\ xs
\end{array}
$$

All of them are used on the RHS to construct a vector so in this sense, each argument of cons is used. However, if we never inspect some fields of (::), or if they always end up in erased positions, or if the whole inspection occurs in an erased part of the program, then the corresponding arguments of cons are erasable – and we would like to recognise them as such. (Often, the erasable fields will be a and n.)

This also means that, unlike in the previous cases, we cannot tell whether an argument of a function is erasable locally, just from its definition, because we also need to know how data constructors are used in the rest of the program. Data constructors thus introduce non-local dependencies.

How it works As we've seen in the example with const^C, whenever projecting a field from a constructor, we need to associate two atoms with this projected variable. For example, consider the following function.

$$
\begin{array}{l}
f = \lambda n : \mathbb{N}.\ \textbf{case } n \textbf{ of} \\
\qquad\qquad\qquad \mathsf{Z} \quad \Rightarrow \ldots \\
\qquad\qquad\qquad \mathsf{S}\ n' \;\Rightarrow \ldots
\end{array}
$$

Above, usage of n' implies that:

- the first argument of function f is used (represented by f_1);
- the field of constructor S is read somewhere in the program (represented by S_1).

Correspondingly, if we have a function $g\ x = \mathsf{S}\ x$, then we need to store x in the field of constructor S only if it is ever read in the program. This is expressed by the implication $g_1 \longleftarrow \{g_\star, \mathsf{S}_1\}$.

4.1.1.6 Composition

Finally, in the following (contrived) example, we compose several different elements, obtaining dependency loops across multiple definitions.

```
data M : Type where
  MkM : ℕ → M

f : (n : ℕ) → (m : M) → M
f  Z     (MkM k)  =  MkM k
f  (S n′) (MkM k)  =  f n′ (MkM (const^A k n′))
```

In the program above, assuming that no other functions read from MkM, the argument m of f can be erased but the analysis is a bit more involved this time.

In the first clause, the argument of MkM, k, is passed to MkM on the RHS and thus it is used if the argument of MkM is used at all – read anywhere in the program.

In the second clause, the first field of MkM, k, is used if const^A uses its first argument, *and* MkM uses its argument (is read anywhere in the program), *and* f uses its second argument.

Since in each case in the program, the usage of the argument of MkM depends on the usage of the argument of MkM, it is consistent to *choose* that the argument of MkM is *not* used.

This then implies that besides the argument of MkM, the argument m of function f can be erased, too, yielding the following program.

```
data M where
  MkM : M

f : (n : ℕ) → M
f  Z     =  MkM
f  (S n′) =  f n′
```

4.1.2 Larger example: binary numbers

Let us have a look at the binary adder (Section 2.2.8.4). I explained in Section 3.1.2.3 that without proper erasure, adding two w-bit numbers takes $O(2^w)$ time (and space). The following illustrates that erasure can recover the linear complexity of the binary adder.

Reminder: the type family of binary numbers Recall from Section 2.2.8.4 that binary numbers are defined as snoc-lists of bits, indexed with their length and the

value that they represent.

```
data Bit : ℕ → Type where
  O : Bit 0
  I : Bit 1

data Bin : (width : ℕ) → (value : ℕ) → Type where
  N   : Bin Z Z
  (#) : {w : ℕ} → {n : ℕ} → {b : ℕ}
        → Bin w n → Bit b → Bin (S w) (b + n + n)
```

In these declarations, I explicitly list all implicit arguments, most importantly the argument n to the constructor (#), which is responsible for the exponential blow-up.

One-bit full adder The function adb sums three bits, producing a two-bit number represented by the type family Sum.

```
data Sum : ℕ → ℕ → ℕ → Type where
  MkS : {hi : ℕ} → {lo : ℕ} → {c : ℕ} → {x : ℕ} → {y : ℕ}
        → Bit hi → Bit lo
        → c + x + y ≡ lo + hi + hi
        → Sum c x y

adb : {c : ℕ} → {x : ℕ} → {y : ℕ} → Bit c → Bit x → Bit y → Sum c x y
adb {1} {1} {1} I I I = MkS {1} {1} {1} {1} {1} I I (Refl {3})
adb {1} {1} {0} I I O = MkS {1} {0} {1} {1} {0} I O (Refl {2})
adb {1} {0} {1} I O I = MkS {1} {0} {1} {0} {1} I O (Refl {2})
...
```

The fully explicit version od adb is extremely verbose but conceptually simple: if we match on the three bits, the rest of each clause is uniquely determined.

If we ensure[1] that the case-tree elaboration inspects the bits before their indices c, x, and y, the inspections of the indices become unnecessary, like in the example with vsum in Section 4.1.1.4. An example of such elaboration can be found in Figure 4.1.

Given the above, the arguments c, x, and y of function adb are erasable. However, note that erasure of these arguments helps only a little since all of them are $O(1)$-sized unary naturals.

Ripple-carry adder The ripple-carry adder adc uses the one-bit full adder adb to sum the bits of two w-bit numbers and uses the following lemma for substitution in

[1]Section 4.5.1.1 discusses how.

```
adb : {c : ℕ} → {x : ℕ} → {y : ℕ} → Bit c → Bit x → Bit y → Sum c x y
adb = λ{c : ℕ}. λ{x : ℕ}. λ{y : ℕ}. λbc : Bit c. λbx : Bit x. λby : Bit y.
  case bc of
    I ⇒ case bx of
          I  ⇒ case by of
                 I  ⇒ case c of
                        1 ⇒ case x of
                              1 ⇒ case y of
                                    1 ⇒ MkS {1} {1} {1} {1} {1} I I (Refl {3})
                 O  ⇒ case c of
                        1 ⇒ case x of
                              1 ⇒ case y of
                                    0 ⇒ MkS {1} {0} {1} {1} {0} I O (Refl {2})
          O  ⇒ case by of
                 I  ⇒ case c of
                        1 ⇒ case x of
                              0 ⇒ case y of
                                    1 ⇒ MkS {1} {0} {1} {0} {1} I O (Refl {2})
                 ...
    ...
```

FIGURE 4.1: An elaborated version of the one-bit full adder

the type.

```
lemmaB : (c + b + b′ ≡ lo + hi + hi)
  → lo + (hi + n + n′) + (hi + n + n′) ≡ c + (b + n + n) + (b′ + n′ + n′)
```

I elaborate the function adc only partially to just expose the handling of the implicits but I will not desugar the case expression, nor elaborate it into case trees, nor fill in the implicit arguments of subst.

```
adc : {c : ℕ} → {w : ℕ} → {m′ : ℕ} → {n′ : ℕ}
    → Bit c → Bin w m′ → Bin w n′ → Bin (S w) (c + m′ + n′)
adc {c} {Z} {Z} {Z} bc N N = N # c
adc {c} {S w} {x + m + m} {y + n + n} bc ((#) w m x xs bx) ((#) w n y ys by) =
  case adb bc bx by of
    MkS {hi} {lo} {c} {x} {y} bhi blo eq ⇒
      subst (Bin (S w)) {_} {_} (lemmaB eq)
        ((#) {S w} {x + m + m + y + n + n} {lo} (adc {hi} {w} {m} {n} bhi xs ys) blo)
```

Having the workhorse function, adc, defined, we can implement the function that adds two binary numbers very easily.

```
addBin : Bin w m → Bin w n → Bin (S w) (m + n)
addBin = adc O
```

In function adc, we can again observe that all variables that stand for natural numbers end up in erasable positions or are entirely unused. In particular, the exponentially large variables, m and n, projected out of the constructor (#), end up in the same position of the constructor (#) or as arguments m' and n' in the recursive call on the RHS, and are therefore unused. Furthermore, the arguments of adc that I named m' and n' here are entirely forced to $x + m + m$ and $y + n + n$ and thus not inspected.

We can therefore see that the arguments m' and n' of function adc and the argument n of constructor (#) are erasable (together with the other indices that we are not worried so much about, such as w), for reasons that I illustrated in Section 4.1.1.

Erasability of argument n of constructor (#) especially means that the expression $x + m + m + y + n + n$ on the RHS need not be evaluated – and that is precisely where the exponentially sized unary numbers would be constructed in each iteration.

Summary In the illustration above, we took a few reasonable assumptions, such as subst using only the last one of its arguments, that the fields of the constructor (#) that are unused in adc are not read elsewhere in the program, and similar.

This allows us to conclude that all expensive computation occurs in erasable positions and we can skip it at runtime.

4.1.3 Compilation process of Idris

The overall process of compiling Idris programs is illustrated in Figure 4.2. The elaboration stage translates the surface language, Idris, into the core calculus, $\mathsf{TT}_{\mathsf{case}}$, by filling in all blanks, omitted types, and expressing high-level language constructs in the core calculus. $\mathsf{TT}_{\mathsf{case}}$ is a dependently typed lambda calculus with case trees as the pattern matching facility.

A program in $\mathsf{TT}_{\mathsf{case}}$ is then stripped of *types*, producing a program in IR, the untyped intermediate representation. We perform erasability analysis on this intermediate representation. The results of erasability analysis inform the erasure transformation, which erases all erasable portions of the given program, yielding a program expressed in $\mathsf{IR}_{\Box}$.

Finally, the program in $\mathsf{IR}_{\Box}$ is processed by the remaining stages of the compiler back end to produce an executable.

4.2 The calculi IR and $\mathsf{IR}_{\Box}$

The erasure approach presented in this chapters analyses programs expressed in the calculus IR, producing *erased* programs expressed in $\mathsf{IR}_{\Box}$. The calculus $\mathsf{IR}_{\Box}$ is identical to IR, except for an added symbol $\Box$, which stands for erased terms. I will write $\mathsf{IR}_{(\Box)}$ to refer to both calculi.

In $\mathsf{IR}_{(\Box)}$, a program is a sequence of function definitions D and data constructor declarations C. The variables v_i stand for formal parameters of the function named n and its body is a term of $\mathsf{IR}_{(\Box)}$.

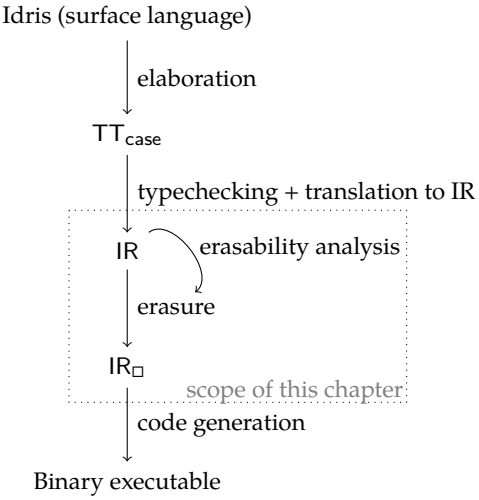


FIGURE 4.2: Idris Compilation Process

$$
\begin{array}{lcl}
D & ::= & \mathsf{n}\ v_1\ v_2\ \ldots\ v_m = T \\
C & ::= & \textbf{constructor}\ \mathsf{n}\ \text{with}\ m\ \text{arguments} \\
T & ::= & \mathsf{c} \mid v \mid \mathsf{n} \\
 & \mid & \square \quad \textit{— only in } IR_{\square} \\
 & \mid & T\ T \mid \lambda v.\ T \\
 & \mid & \textbf{let}\ v = T\ \textbf{in}\ T \\
 & \mid & \left\{ \begin{array}{l} \textbf{case}\ T\ \textbf{of} \\ \quad \mathsf{n}^1\ v_1^1\ v_2^1\ \ldots\ v_{m^1}^1 \Rightarrow T \\ \quad \mathsf{n}^2\ v_1^2\ v_2^2\ \ldots\ v_{m^2}^2 \Rightarrow T \\ \quad \vdots \\ \quad \mathsf{n}^k\ v_1^k\ v_2^k\ \ldots\ v_{m^k}^k \Rightarrow T \end{array} \right.
\end{array}
\tag{4.1}
$$

FIGURE 4.3: Term syntax of the intermediate representations $\mathsf{IR}_{(\square)}$.

An $\mathsf{IR}_{(\square)}$ term T is either a constant, a variable, a global name; an erased term; an application, a lambda; a let-binding; or a case-expression, whose individual branches refer to data constructors n_i with their fields being pattern-matched as v_i^j.

We distinguish (local) names of variables v from global names of definitions n.

The language $\mathsf{IR}_{(\square)}$ is untyped. This allows us to remove parts of the program arbitrarily, without restrictions (but also guarantees) of any typing constraints that would be invalidated by erasure.

4.3 Erasure inference

Given a program expressed in IR, we can infer erasability from it by formalising the intuition given in Section 4.1.

4.3.1 Basic notions

Let us define the following terms.

Usage An item is *used* if it may influence the normal form of main. Otherwise it is *unused*.

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.

Arity of a global name The number of Π binders present literally at the beginning of the type signature of the global name.

I define this because the arity of a function may not be clear in variadic functions like printf. In this scheme, variadic functions are modelled as fixed-arity functions returning other functions, where the "other" functions are not subject to erasure.

Node A pair of a global name n and argument number i, usually written n_i. A node stands for a position that can be considered used/erased or not.

We also introduce node $f_\star$ for every function f, to denote the return value of f (which also may or may not be used).

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 its set of dependencies must be considered used as well.

Guards A set of nodes whose simultaneous usage implies that some other node is 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".

Usage pattern A mapping from a set of nodes to the set {possibly used, unused}.

4.3.2 Implication gathering

In order to infer erasability, the algorithm searches the call graph to discover which functions are (transitively) referred to from main. From each function definition discovered, it collects a set of *implications* describing how its arguments are used.

Implications are generated by the operation $[\![-]\!]^\Gamma_G$, which takes a term and returns a set of implications. The indices G and Γ describe the current context. G is a set of *guards* for the current context and Γ is an environment that maps variable names to their dependency sets, as described further below. Figure 4.4a contains the implication-generating rules for definitions and terms; Figure 4.4b describes how to generate implications for case expressions. Constructor declarations do not generate implications.

4.3.2.1 Function definitions

Preprocessing Before analysis, we η-expand function definitions, wherever applicable, generating fresh names for the new arguments. Any name defined with a functional type should have the corresponding number of lambdas at the beginning of its definition.

The η-expansion procedure uses arity as defined in Section 4.3.1.

Implications Equation 4.2 in Figure 4.4a describes how to gather implications from function definitions. The body of f, written B_f, is analysed as a term in the environment and with the guards given in the equation.

Guards We start analysing the body of function f with the set of guards containing $f_\star$. This expresses that whatever usage is inferred from the body of f, it should be considered only if the return value of f is used at all.

Environment of dependencies The environment 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 environment 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 $\Gamma(x)$ to denote the set of dependencies (nodes) assigned to x. A variable may draw in a non-singleton set of dependencies if it arises from nested pattern matching, which is represented in $\mathsf{IR}_{(\square)}$ by nested case expressions (case 4.4 in Figure 4.4b).

Example Let us look at the following function as an example.

$$\begin{aligned}&\mathsf{makeEven} : \mathbb{N} \to \mathbb{N}\\&\mathsf{makeEven} = \mathsf{double} \circ \mathsf{halve}\end{aligned}$$

First, we η-expand to $\mathsf{makeEven}'\ x = (\mathsf{double} \circ \mathsf{halve})\ x$. Then the variable x is the first formal argument of the function $\mathsf{makeEven}'$ and it is represented by the node $\mathsf{makeEven}_1$. The initial dependency environment Γ is $\{x \mapsto \{\mathsf{makeEven}_1\}\}$, and the initial set of guards G is $\{\mathsf{makeEven}_\star\}$.

4.3.2.2 Terms

The remaining equations in Figure 4.4a show how to build implication sets for terms of IR.

- Constants, global names, and local variables;
- applications of global names;
- application of terms that are not global names;

$$[\![f\ v_1\ v_2\ \ldots\ v_m = B_f]\!] \;=\; [\![B_f]\!]^{\{v_1 \mapsto \{f_1\}, \ldots, v_m \mapsto \{f_m\}\}}_{\{f_\star\}} \tag{4.2}$$

$$\begin{aligned}
[\![\mathsf{c}]\!]^\Gamma_G &= \emptyset && \text{— constant}\\
[\![\mathsf{n}]\!]^\Gamma_G &= \{\mathsf{n}_\star \longleftarrow G\} && \text{— global name}\\
[\![v]\!]^\Gamma_G &= \{d_i \longleftarrow G \mid d_i \in \Gamma(v)\} && \text{— variable}\\
[\![\mathsf{n}\ T_1\ T_2\ \ldots\ T_m]\!]^\Gamma_G &= \{\mathsf{n}_\star \longleftarrow G\} \cup \textstyle\bigcup_{i=1}^m [\![T_i]\!]^\Gamma_{\{\mathsf{n}_i\} \cup G}\\
[\![F\ T_1\ T_2\ \ldots\ T_m]\!]^\Gamma_G &= [\![F]\!]^\Gamma_G \cup \textstyle\bigcup_{i=1}^m [\![T_i]\!]^\Gamma_G\\
[\![\textbf{let}\ v = T\ \textbf{in}\ M]\!]^\Gamma_G &= [\![T]\!]^\Gamma_G \cup [\![M]\!]^{\Gamma \cup \{v \mapsto \emptyset\}}_G\\
[\![\lambda v.\ M]\!]^\Gamma_G &= [\![M]\!]^{\Gamma \cup \{v \mapsto \emptyset\}}_G
\end{aligned}$$

(A) Implication gathering: function definitions and terms

$$\left[\!\!\left[\begin{array}{l}\textbf{case}\ T\ \textbf{of}\\ \quad \mathsf{n}^1\ v_1^1\ v_2^1\ \ldots\ v_{m^1}^1 \;\Rightarrow\; B^1\\ \quad \mathsf{n}^2\ v_1^2\ v_2^2\ \ldots\ v_{m^2}^2 \;\Rightarrow\; B^2\\ \qquad\qquad \vdots\\ \quad \mathsf{n}^k\ v_1^k\ v_2^k\ \ldots\ v_{m^k}^k \;\Rightarrow\; B^k\end{array}\right]\!\!\right]^\Gamma_G = [\![T]\!]^\Gamma_G \cup \bigcup_{i=1}^{k} [\![B^i]\!]^{\Gamma \cup V^i}_G \tag{4.3}$$

where $V^i = \left\{v^i_j \mapsto \left\{\mathsf{n}^i_j\right\} \;\middle|\; j \in 1 \ldots m^i\right\}$
and ($k \neq 1$ or T is not a variable)

$$\left[\!\!\left[\begin{array}{l}\textbf{case}\ v\ \textbf{of}\\ \quad \mathsf{n}\ v_1\ v_2\ \ldots\ v_m \;\Rightarrow\; B\end{array}\right]\!\!\right]^\Gamma_G = [\![B]\!]^{\Gamma \cup V}_G \tag{4.4}$$

where $V = \left\{v_j \mapsto \left(\{\mathsf{n}_j\} \cup \Gamma(v)\right) \;\middle|\; j \in 1 \ldots m\right\}$

(B) Implication gathering: case expressions

- let expressions;
- lambda expressions.

Constants and references Constants do not generate any constraints.

Global names, in a context represented by the set of guards G, produce one implication, $\mathsf{n}_\star \longleftarrow G$. This means that in the context guarded by G, the (return) value of n is used.

Local variables do not have a dedicated node $-_\star$ like global names. Instead, they have dependency sets recorded in the environment Γ. This environment keeps track of where the local variable comes from, whether it comes from a function argument, a pattern variable in a case match, or a sequence of single-branch case matches.

If a local variable v is used in the context represented by guards G, all nodes from its dependency set must be used, too. This is expressed by the set of implications $\{d_i \longleftarrow G \mid d_i \in \Gamma(v)\}$.

In general, tautologies can be dropped and implications can be optimised at this point. For example, the above set can be reduced to $\{d_i \longleftarrow G \mid d_i \in \Gamma(v) \setminus G\}$.

Applications of global names Applications of global names are the key element in our erasure algorithm. Erasure corresponds to pruning an abstract syntax tree just below the nodes corresponding to applications of globals, removing some of their subterms.

To explain the rules of usage in applications of globals, consider when the variable x is used in the body of f in the following definition.

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

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. Even if g_2 is used, h may not use *its* first argument, which would also make x unused again. In other cases, we mark x as used.

As we have seen, the usage of x depends on g_2 and h_1, 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 $[\![-]\!]^\Gamma_G$.

We can generalise the above example and write a general rule for application of global names (i.e. functions or data constructors), as shown in Figure 4.4a. For every argument T_i, we include its set of implications, guarded by f_i. This precondition added to G expresses that these implications come into effect only if the i-th argument of f is used.

Finally, each application of a global name n uses the return value of n, which yields an extra implication $\mathsf{n}_\star \longleftarrow G$.

Applications of non-globals 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 anything else is that we extend the set of guards G in the operands when applying globals.

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.

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.

Let expressions Let-bound values are considered used, regardless of how (and whether at all) the bound variable is used on the RHS of the let expression. For the RHS M, the environment Γ is extended with the empty dependency set for v because the associated set of implications $[\![T]\!]^{\Gamma}_{G}$ has already been unconditionally added to the constraint set.

In practice, this has led to surprising effects where a programmer complained about insufficient erasure when binding proofs using **let**, and that moving the proofs into a **where** block fixed the issue since in Idris, **where** definitions are lifted to top level.

A way to solve this problem would be generating a unique global name for each let-bound definition, and treating them like global definitions in the erasure mechanism. This would simulate lifting let-bound definitions to the global scope. It has not been implemented in Idris 1.

Another way would be not including the implications from T immediately but associating them with v in Γ instead. This would however require a more complex structure of dependency sets in Γ (Section 4.5.4) than we have now.

Lambda expressions From the point of view of erasure and usage, lambda expressions are delayed computation. Since this erasure scheme cannot track control flow, we have to assume that all resources in a lambda expression are used at the point of creating of the anonymous function.

Furthermore, since the rule for applying non-globals, which includes lambdas and higher-order variables, unconditionally includes dependencies from the argument, there are no dependencies we need to (and possibly could) assign to the variable v, and thus we extend the environment Γ with $v \mapsto \emptyset$.

4.3.2.3 Case expressions

Analysis of the most general form of case expressions is shown in the first equation in Figure 4.4b. The inspected term T is certainly used and therefore T, together with each branch of the case expression, contributes to its set of implications.

In each branch, the context Γ is extended with V^i, which describes the new variables introduced by pattern matching. V^i expresses that the value v^i_j is obtained by reading the j-th field of the constructor n^i, represented by the node n^i_j.

Single-branch case expressions In Figure 4.4b, I include a rule for 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.

If all projections end up in erased positions or are not used at all (for illustration, this usually happens when matching on Refl, the constructor of propositional equality), we want to avoid marking the inspected variable as used.

In such cases, the erasure step *must* later remove case-inspections of erased variables in order to keep the inspected expression unevaluated.

This optimisation is intended for forced constructors (cf. Section 7.2.3.2, rule 7.3) and does not preserve pattern match failures if the pattern is not covering.

Pushing case inspection into the leaves This single-branch optimisation could also be achieved by "pushing case inspections into the leaves" using the following rewrite rule. This has to be done *before* erasure analysis.

$$\begin{array}{l}\textbf{case } v \textbf{ of} \\ \quad \mathsf{n}\ v_1 \ldots v_m \Rightarrow R\end{array} \quad \Longrightarrow \quad R \left[v_i \mapsto \begin{array}{l}\textbf{case } v \textbf{ of} \\ \quad \mathsf{n}\ v_1 \ldots v_m \Rightarrow v_i\end{array} \right]_{i \in 1..m}$$

Above, $[v_i \mapsto \ldots]_{i \in 1..m}$ stands for substitution for variables v_1 through v_m.

If all references to a variable are unused in the original program, all copies of its corresponding case expression will be unused and erasable in the transformed program, too.

However, depending on the implementation of the language, this transformation may or may not incur runtime cost because it copies (constantly sized) pieces of code.

IR, as implemented in Idris 1, has a special term form called $\mathsf{ProjCase}_i(T)$ that projects the i-th field from the constructor at the head of the normal form of T. This term form is used for optimisations; it is low-level and unsafe and it does not check which constructor it is projecting from, whether the index i is in bounds, or whether the piece of memory in question represents a constructor at all.

We can view ProjCase as a shorthand for a case expression that has been "pushed into the leaf". Indeed, if we extend ProjCase to store the name of the constructor n, the difference between ProjCase and a case expression pushed into the leaf reduces to just syntax.

Single-branch inspection of terms The single-branch optimisation in Figure 4.4b is restricted to the cases where the scrutinee is a variable. Therefore, case inspections of compound terms will not be recognised as redundant, even if there is only one branch and all variables projected from the match are unused.

This is sufficient for Idris 1 because its case-tree elaboration algorithm [Wad87a] produces only case trees that inspect variables.

We could extend the single-branch optimisation for singleton case expressions inspecting *compound terms* but that would require a more elaborate formulation of dependency sets than we currently have (Section 4.5.4).

4.3.3 Dependency solving

Interpretation as a logic program After collecting a set of implications from the program, the algorithm uses it to find the minimal consistent usage pattern of the program. We interpret every node as a propositional variable, a set of guards as a conjunction of variables, and every implication as logical implication.

Intuitively, the propositional variable f_i is true if the function f uses its i-th argument (when f is a function), or if the data constructor f needs to store its i-th argument (when f is a data constructor) in order to preserve the result of main.

With this interpretation, each implication collected by the program is a Horn clause, e.g.:

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

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 order to stay erasure-consistent.

Therefore, the output of implication gathering can be seen as a (0-th order) logic program.

Postulates We extend the set of implications gathered from the program with the following implication.

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

This implication has an empty set of guards on the RHS and it expresses that the return value of main is used unconditionally.

In the presence of compiler primitives or other foreign definitions, we may have to add additional postulates, as described in Section 4.5.2.

Finding the minimal solution Let Δ be the set of Horn clauses gathered from the program, including all postulates and other extra alements added to it. Then the minimal solution is the smallest set S such that the following rule holds.

$$\frac{G \subseteq S \qquad (n_i \longleftarrow G) \in \Delta}{n_i \in S}\ \textsc{ForwardChain}$$

The set S is the minimal model of the logic program Δ.

$$
\begin{aligned}
&\langle \mathsf{c} \rangle_S && = \mathsf{c} \\
&\langle v \rangle_S && = v \\
&\langle \mathsf{n} \rangle_S && = \mathsf{n} \\
&\langle \lambda v.\ T \rangle_S && = \lambda v.\ \langle T \rangle_S \\
&\langle \textbf{let } v = T \textbf{ in } M \rangle_S && = \textbf{let } v = \langle T \rangle_S \textbf{ in } \langle M \rangle_S \\
&\langle F\ T_1 \ldots T_m \rangle_S && = \langle F \rangle_S\ \langle T_1 \rangle_S \ldots \langle T_m \rangle_S \\
&\quad \textbf{where} \\
&\quad F \text{ is neither a global name, nor an application}
\end{aligned}
\tag{4.5}
$$

$$
\begin{aligned}
&\langle \mathsf{n}\ T_1\ T_2\ \ldots\ T_m \rangle_S\ =\ \mathsf{n}\ T'_1\ \ldots\ T'_m \\
&\quad \textbf{where} \\
&\quad \mathsf{n} \text{ is a (global) name of a function} \\
&\quad T'_i\ =\ \begin{cases} \square & \text{if } (\mathsf{n}_i \notin S) \wedge \left(i < \textsc{Arity}(\mathsf{n})\right) \\ \langle T_i \rangle_S & \text{otherwise} \end{cases}
\end{aligned}
\tag{4.6}
$$

$$
\begin{aligned}
&\langle \mathsf{n}\ T_1\ T_2\ \ldots\ T_m \rangle_S := \langle \mathsf{n} \rangle^k\ \langle T_{a_1} \rangle_S\ \ldots\ \left\langle T_{a_k} \right\rangle_S \\
&\quad \textbf{where} \\
&\quad \mathsf{n} \text{ is a name of a constructor} \\
&\quad \{a_i\}_{i=1}^{k}\ =\ \text{maximal subsequence of } 1 \ldots m \text{ where } \forall i.\ \mathsf{n}_{a_i} \in S \\
&\quad \langle \mathsf{n} \rangle^k\ =\ \text{a variant of } \mathsf{n} \text{ restricted to } k \text{ fields}
\end{aligned}
\tag{4.7}
$$

FIGURE 4.5: Erasure from terms

Since Δ describes the data dependencies within the program, then nodes in S must be considered used and unerasable. For the nodes outside S, there is no reason to consider them used and they can be erased.

The dependency solving stage is discussed more broadly in Chapter 6. Especially, its efficiency is discussed in Section 6.5.2.

4.4 Erasure

The removal of non-computational code happens by pruning the AST of a program, which transforms a program expressed in IR to a program expressed in $\mathsf{IR}_\square$. I represent erasure by the operation $\langle - \rangle_S$, defined in Figures 4.5 and 4.6.

4.4.1 Terms

In Figure 4.5, Equation 4.5 shows how to prune terms like constants, variables, global names, applications, lambdas and let-expressions. These do not change beyond being pruned recursively.

Applications of global functions In Figure 4.5, Equation 4.6 shows that we prune the syntax tree by replacing erased positions in applications of global functions with athe placeholder □.

This approach does not change arities of functions; it does not remove formal arguments from functions. Instead, we just do not perform the erased pieces of computation and leave the corresponding references undefined[2] in the low-level representation.

The (meta-) function Arity computes arity as defined in Section 4.3.1 – by counting the literal Π binders in the type signature of n.

Data constructor applications In a way, data constructors are simpler than functions because they have more clearly defined arities, and other parts of the Idris compiler keep them exactly saturated in the internal representation. Therefore, it is easier to completely *remove* their fields, changing their arities, instead of filling them with placeholders (□). This is shown in Figure 4.5, Equation 4.7.

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.

For example, consider the following program, where we will assume that the argument z of constructor C can be erased.

```
data T : Type where
  C : (x : ℕ) → (y : ℕ) → (z : ℕ) → T

x : (Bool, ℕ → ℕ → T)
x = (False, C 3)
```

Then the definition x must erase to the following.

$$x = (\mathsf{False}, \lambda y.\ \lambda z.\ \langle \mathsf{C} \rangle^2\ 3\ y)$$

We had to η-expand the application of C because the erasure scheme in this chapter does not support erasure in non-global functions. Hence any code that receives the function (C 3) will treat it as a function with type $\mathbb{N} \to \mathbb{N} \to \mathsf{T}$ with no erased arguments and will attempt to apply it to (up to) two arguments.

Since we do remove the arguments in the low-level representation of C but we cannot propagate that fact with the resulting higher-order function, we must use η-expansion to create a low-level interface that matches the type of the function.

4.4.2 Case expressions

Case expressions are erased as described in Figure 4.6. We recursively erase all involved parts:

- the inspected term;

[2] In the C code generator of Idris 1, these references are set to NULL.

$$\left\langle \begin{array}{l} \textbf{case } v \textbf{ of} \\ \quad \mathsf{n}\ v_1\ v_2\ \ldots\ v_m\ \Rightarrow\ B \end{array} \right\rangle_S = \langle B \rangle_S \quad \textbf{if } \forall i.\ v_i \text{ not free in } \langle B \rangle_S \tag{4.8}$$

$$\left\langle \begin{array}{l} \textbf{case } T \textbf{ of} \\ \quad \mathsf{n}^1\ v_1^1\ v_2^1\ \ldots\ v_{m^1}^1 \Rightarrow B^1 \\ \quad \mathsf{n}^2\ v_1^2\ v_2^2\ \ldots\ v_{m^2}^2 \Rightarrow B^2 \\ \quad \vdots \\ \quad \mathsf{n}^k\ v_1^k\ v_2^k\ \ldots\ v_{m^k}^k \Rightarrow B^k \end{array} \right\rangle_S = \begin{array}{l} \textbf{case } \langle T \rangle_S \textbf{ of} \\ \quad \left\langle \mathsf{n}^1\ v_1^1\ v_2^1\ \ldots\ v_{m^1}^1 \right\rangle_S \Rightarrow \langle B^1 \rangle_S \\ \quad \left\langle \mathsf{n}^2\ v_1^2\ v_2^2\ \ldots\ v_{m^2}^2 \right\rangle_S \Rightarrow \langle B^2 \rangle_S \\ \quad \vdots \\ \quad \left\langle \mathsf{n}^k\ v_1^k\ v_2^k\ \ldots\ v_{m^k}^k \right\rangle_S \Rightarrow \langle B^k \rangle_S \end{array} \quad \textbf{if previous rule (4.8) does not apply} \tag{4.9}$$

FIGURE 4.6: Erasure from case expressions

- right-hand sides of each branch;
- left-hand sides of each branch.

The left-hand sides are erased using the rule described in Figure 4.5, Equation 4.7, because at runtime, the arities of data constructors might be different.

If we use the singleton-case optimisation from Figure 4.4b, Equation 4.4, we must also remove unused case splits according to Figure 4.6, Equation 4.8, to ensure that the scrutinee is not inspected at runtime, as the corresponding low-level reference may be undefined.

The resulting erased program is now free from all unused data that we have identified, and is ready to be compiled further by the back-end to an executable.

4.5 Extensions

In a practical language implementation, there are additional language elements that interact with erasure, such as type classes or compiler primitives. We may also wish to add features for better ergonomy of the erasure mechanism, such as explicit erasure annotations and more helpful erasure warnings. This section discusses a few such options.

4.5.1 Erasure annotations

My implementation in Idris 1 allows user-provided explicit erasure annotations. Perhaps unintuitively, they cannot be used to request erasure of the annotated elements. Instead, they influence erasure only indirectly, via the second of their two purposes:

```
data Vect : ℕ → Type → Type where
  Nil  :  Vect Z a
  (::) :  .{n : ℕ} → (x : a)
          → (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
```
(4.10)

FIGURE 4.7: Syntax of erasure annotations in Idris 1

- requesting warnings if any annotated part of the program is *not* erased;
- influencing case-tree elaboration to prefer inspecting unannotated values.

In Idris 1, 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 4.7, 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. They can improve it indirectly via case tree elaboration, but only because Idris 1 does not have other facilities to fully specify the desired runtime behaviour (see Section 4.5.1.1 below).

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

4.5.1.1 Case-tree elaboration

When elaborating pattern-matching function definitions into case trees, there may be several choices, as already discussed in Section 2.1.4.2.

For example, the function halve could perform computation on either of its two arguments, erasing the other.

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

Both versions would work *correctly* but they would have different erasure properties, and thus different runtime behaviour, possibly with even asymptotically different

runtimes and/or memory usage. This is especially important because erasure cascades.

In Section 2.1.4.3, I argue that the choice of the interpretation is up to the programmer. However, Idris 1 does not have a way for programmers to choose which elaboration they desire because there is no facility to mark forced patterns.

This is partly because Idris's former erasure scheme (forcing, collapsing, and detagging, Section 3.2.1.1) used to make the choice for the programmer by always inspecting function arguments instead of constructor fields, wherever possible. Idris also uses other heuristics to build efficient case trees, such as inspecting the variable with the highest number of distinct constructors first, or preferring to split "from the right", assuming that indices would be mostly forced by the patterns to the right of them.

In such cases, the user can influence the case-tree elaborator by putting an explicit erasure annotation in the type of the function or in types of data constructors and the case-tree elaborator will try to avoid inspecting dotted arguments, if possible.

The algorithm The compiler of Idris uses a variant of the pattern compiling algorithm described by Wadler [Wad87a], with its own modifications and heuristics, some of them described in the paragraphs above.

I modified the pattern compiler 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.

This is achieved by modifying Wadler's function match to sort its list of variables, together with their corresponding patterns, by whether they are dotted or not (besides the other criteria that Idris used before).

In practice, this amounts to sorting the list of variables and patterns at the beginning of the transformation and then every time Wadler's constructor rule is applied. In other words, the list of pattern variables (and their corresponding patterns) is sorted every time new pattern variables are added to it.

4.5.1.2 Free implicits

As mentioned in Section 2.1.1, free implicits are free variables occurring in type signatures. In the type signature below, a is a *free implicit* and n is a *bound implicit*.

$$\mathsf{vlen} : \{n : \mathbb{N}\} \to \mathsf{Vect}\ n\ a \to \mathbb{N}$$

Free implicits are silently bound by Idris at the beginning of the type signature with their inferred type.

In Idris 1, we also chose to make free implicits "dotted" (erasure-annotated) by default. This is a compromise solution to make the pattern compiler choose the interpretation "that the user meant" without making the user to put in too many

erasure annotations. In practice, we have found it consistent with the usual intention and it provides extra checks.

However, it is still a compromise and a mild case of the Hindley-Milner conflation (Section 3.2.2.2), which here extends to the fact that the same facility – erasure annotations – are used to control two entirely different things, *erasure* warnings, and *elaboration* of case trees.

This trick could be removed if Idris 1 gets another way to specify the desired operational behaviour of pattern matching definitions.

4.5.1.3 Erasure warnings

After usage analysis, the compiler checks whether everything that has been dotted (marked with erasure annotations) is also recognised as erasable. If that is not the case, it issues a warning for each case where an erasure annotation is violated.

Erasure warnings are, however, 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.

Erasure warnings are discussed further in Section 4.5.6.

4.5.2 Primitives and builtins

The erasure inference algorithm (Section 4.3) generates implications only from in-language function definitions. However, all external definitions, foreign calls and compiler builtins have no in-language definitions and they would therefore end up recognised as not using any of their arguments.

Therefore we introduce *usage postulates*, which assert which arguments of which builtins are used. In the case of Idris 1, some usage postulates are built into the compiler, and they can also be introduced by a programmer with a compiler pragma.

For example, the fact that the return value of main is used, is also implemented as a usage postulate in Idris.

$$\mathsf{main}_\star \longleftarrow \emptyset \tag{4.11}$$

We saw this postulate in Section 4.3.3 on dependency solving.

4.5.3 Type classes

The erasure scheme, as presented, cannot express higher-order erasure patterns, such as data constructors taking functions with erased arguments. (This is further discussed in Section 4.5.5.)

However, type classes are an important higher-order structure, since they are implemented as records containing functional fields, and we would like to erase from methods of type classes as if they were normal functions.

Type class methods usualy do not have (immediate) definitions and therefore erasure inference would not infer any usage for them. We thus define the usage pattern of a method as the *union* of usage of all its instances, which is the conservative approximation. More precisely, we define the usage of a method as the union of the usage of every function passed to the instance constructor anywhere in the program.

For example, this will likely compute the usage of Num.(+) as the union of the usage of $+_{\mathbb{N}}$, $+_{\mathsf{Int}}$, $+_{\mathsf{Double}}$, etc. In Idris, this may include runtime-constructed instances.

In Idris 1, I implemented special support for type classes as given in the following description.

- We generate a fresh global name for each typeclass method. In the following, let us assume that the name is n^j for each instance constructor n and number of method j.
- We extend the environment Γ in Figures 4.4a and 4.4b to track which variables represent which typeclass methods.
- We extend usage analysis in Figure 4.4a with the following rule that takes precedence over the general rule for applications.

  $$[\![v\ T_1\ T_2 \ldots T_m]\!]^\Gamma_G = \{\mathsf{n}^j_\star \longleftarrow G\} \cup \bigcup_{i=1}^m [\![T_i]\!]^\Gamma_{\{\mathsf{n}^j_i\} \cup G}$$

  if Γ indicates that v represents typeclass method n^j

  Without this rule, all arguments T_i would be analysed as used in the context represented by G. With this rule, usage of every argument T_i has an extra precondition n^j_i that extends G in the recursive application of $[\![\cdot]\!]$.

- We further extend usage analysis in Figure 4.4a with the following rule that takes precedence over the general rule for global names.

  $$\begin{aligned}[\![\mathsf{n}\ T_1\ T_2 \ldots T_m]\!]^\Gamma_G \;=\; &\{\mathsf{n}_\star \longleftarrow G\} \cup \bigcup_{i=1}^m [\![T_i]\!]^\Gamma_{\{\mathsf{n}_i\} \cup G} \\ &\cup \bigcup_{j=1}^m [\![\mathsf{n}^j = T_j]\!]\end{aligned}$$

  if n is a typeclass instance constructor

  In the rule above, $[\![\mathsf{n}^j = T_j]\!]$ invokes Equation 4.2 from Figure 4.4a – we analyse the program as if it contained $\mathsf{n}^j = T_j$ as a definition.

  Without this rule, no implications would be generated for the instance names n^j. With this rule, each term T_j passed as the j-th argument to the instance constructor n anywhere in the program also contributes the appropriate erasure implications to n^j.

Because the analysis of function definitions performs η-expansion, this approach works even in Idris, where T_i is always just a global name.

Note that implications for n^j are generated even if G represents an erasable context. This could be fixed easily by passing G down to the analysis of $\mathsf{n}^j = T_j$ or by other means, such as a better representation of dependency sets (Section 4.5.4).

- We extend erasure in Figure 4.6 to keep track of which variables are projected out from a typeclass instance constructor. This amounts to keeping an environment while performing recursion.
- We extend erasure in Figure 4.5 with the following rule that takes precedence over the general rule for applications (which includes applications of local variables).

$$\langle v\ T_1\ T_2\ \ldots\ T_m \rangle_S\ =\ v\ T'_1\ \ldots\ T'_m$$

where

v represents method n^j

$$T'_i\ =\ \begin{cases} \square & \text{if } (\mathsf{n}^j_i \notin S) \wedge \left(i < \textsc{Arity}(\mathsf{n}^j)\right) \\ \langle T_i \rangle_S & \text{otherwise} \end{cases}$$

This means that erasure-wise, v is considered to represent n^j but the generated code still contains v.

Creating names of the form n^j_i is a special case of the more general approach discussed in Section 4.5.5.

4.5.4 Different representation of dependency sets

In Figures 4.4a and 4.4b, we keep Γ, an environment that tracks dependency sets for each variable. Dependency sets are called *dependency* sets because they contain just the dependencies, nodes, as illustrated by the following example.

$$\Gamma(v) = \{\mathsf{n}_2, \mathsf{m}_1\}$$

As we have seen earlier in this chapter, this simplicity precludes more advanced erasure inference, such as better analysis of let bindings (Section 4.3.2.2), single-branch case-inspection of complex terms (Section 4.3.2.3), or better approximation of usage patterns of class methods (Section 4.5.3).

In Chapter 6, we associate a whole *constraint set* with a variable. This set does not contain just nodes that are used immediately when a variable is used, but also nodes that are used *given some precondition* when the variable is used, as illustrated by the following example.

$$\Gamma(v) = \left\{\{\mathsf{p}_3, \mathsf{q}_1\} \rightarrow \mathsf{n}_2, \emptyset \rightarrow \mathsf{m}_1\right\}$$

This approach is more flexible and composable.

For illustration, Equation 4.4 in Figure 4.4b for single-branch case trees could be generalised to support non-variable scrutinees as follows (assuming that the rest of the erasure mechanism has been adapted to the new form of Γ).

$$\left[\!\!\left[\begin{array}{l} \textbf{case } T \textbf{ of} \\ \quad \mathsf{n}\ v_1\ v_2\ \ldots\ v_m\ \Rightarrow\ B \end{array} \right]\!\!\right]^{\Gamma}_{G} = [\![B]\!]^{\Gamma \cup V}_{G}$$

$$\text{where } V = \left\{ v_j \mapsto \left(\{\emptyset \to \mathsf{n}_j\} \cup [\![T]\!]^{\Gamma}_{G} \right) \;\middle|\; j \in 1 \ldots m \right\}$$

The most important difference to the original rule is that the extension to Γ, which we call V, is now free to use $[\![T]\!]^{\Gamma}_{G}$ instead of $\Gamma(v)$.

A similar inference rule would be applicable for let expressions, together with an erasure rule that removes a let-bound definition if it is not referenced from the erased subterm, exactly like with singleton case trees.

4.5.5 Higher-order functions

The erasure mechanism presented in this chapter does not support higher-order functions in the sense that it is unable to express erasure from functional arguments of functions and data constructors.

As an example, consider the following function expressed in the continuation-passing style.

$$f : \left(g : (n : \mathbb{N}) \to \mathsf{Vect}\ n\ \mathsf{Bool} \to a\right) \to a$$

Here, the argument n of g cannot be erased because g does not have a global name and therefore there is no way to create nodes for it in the current scheme.

An example of this problem are also typeclass instances, which are represented in Idris as records with functional fields, as discussed in Section 4.5.3.

I present several suggestions, of which only the special case for type classes is implemented in Idris 1 (Section 4.5.3).

Generalised nodes We could generalise the node naming scheme. Instead of generating nodes $\mathsf{n}_\star$, $\mathsf{n}_1 \ldots \mathsf{n}_m$ for any global name n with arity m, we could use numeric sequences in the subscript of n.

- n, with an empty sequence in the subscript, represents the return value of n
- n_i represents the i-th argument of n
- n_{ij} represents the j-th argument of the i-th argument of n
- n_{ijk} represents the k-th argument of the j-th argument of the i-th argument of n
- ...

This would allow erasure analysis of names that are not global.

Arbitrarily numbered nodes Alternatively, we could just assign a unique integer to all binders in the program, instead of the currently used descriptive name n_i. This is the approach taken in Chapter 5.

This would also allow analysis of lambdas and let expressions, which would otherwise not be assigned nodes even with the generalised-nodes approach above.

Lambda/let lifting An alternative to numbering might be lambda/let lifting, which moves local definitions to the top level, creating a global name for each one before erasure analysis. Idris 1 currently does not perform this before erasure analysis (although it does at code generation time).

Lambda/let lifting would however not help with the function f given at the beginning of this section.

Shortcomings Even the numbered-nodes approach would not be sufficient for the full strength of dependent types. Consider the function f defined as follows.

$$
\begin{aligned}
&\mathsf{fTy} : \mathsf{Bool} \to \mathsf{Type} \\
&\mathsf{fTy}\ \ \mathsf{True}\ \ = \mathbb{N} \to \mathbb{N} \\
&\mathsf{fTy}\ \ \mathsf{False}\ = \mathbb{N} \to \mathbb{N}
\end{aligned}
$$

$$
\begin{aligned}
&f : (b : \mathsf{Bool}) \to \mathsf{fTy}\ b \\
&f\ \ \mathsf{True}\ \ = (\lambda x.\ x) \\
&f\ \ \mathsf{False}\ = (\lambda x.\ 42)
\end{aligned}
$$

In general, this method falls short when the type of a function is itself computed because the definition of arity, and therefore what a formal argument is, and therefore how we generate erasure nodes, relies on explicit Π bindings at the beginning of a type (Section 4.3.1).

It is not clear how to extend this numbering scheme to computed types without considering the types themselves, which this chapter does not. I address this in Chapter 5, which does consider types fully.

4.5.6 Better error reporting

If the programmer annotates any function arguments with explicit erasure annotations (Section 4.5.1), or if the compiler erasure-annotates free implicits by default (Section 4.5.1.2), the compiler checks that the explicit annotations match the inferred erasure pattern.

If a discrepancy is found, the compiler reports an error or a warning. However, in the erasure scheme presented in this chapter, all that the compiler can report is which nodes that have been annotated (asserted to be unused) but found used. Most

importantly, it cannot report *where* they are used, which is the most useful piece information for the programmer[3]

The information where nodes are used is lost because the process of implication gathering and solving them separately is decoupled from the program.

Reasons My implementation in the Idris 1 compiler therefore uses an extended representation of implications.

$$\mathsf{n}_i \xleftarrow{\rho} \mathsf{m}_j, \mathsf{z}_k$$

In the implication above, ρ is the *reason* explaining why the implication exists. The reason usually carries information like "case-inspected as i-th argument of function f".

Eventually, the solver emits the corresponding reasons for each node that has been resolved as used and unerasable. If any of these nodes are erasure-annotated, the user is presented with a list of reasons for each such node.

4.5.7 Newtype optimisation

The newtype optimisation [Jon03] deals with type families with a single constructor that has a single field. Values of such types can be represented at runtime by just the value stored inside, without the constructor wrapper.

In strict languages, like Idris, we are free to perform the newtype optimisation on any single-constructor-single-field type family, since it will not lead to undesired loss of laziness.

Idris does perform the newtype optimisation, and it performs it *after* erasure. With erasure, many complex-typed structures, especially proof-wrapped (for correctness) primitives (for performance), are stripped of types and proofs, and they collapse down to their bare bones low-level representation.

4.6 Results

As simple as it is, the erasure system presented in this chapter can erase all undesired data in all examples presented in Section 3.1.2. A summary of its shortcomings can be found in Section 4.7.1.

4.6.1 Benchmarks

The presented algorithm is implemented in the Idris 1 compiler, and it is used to perform erasure on all programs. In this section, I show how it performs in practice.

[3]Consider the helpfulness of a message like "$\mathsf{Prelude.Vector.(::)}_2$ is annotated but used somewhere in your program".

4.6.1.1 Benchmarked programs

I show the performance of the following three programs[4], introduced in Section 3.1.2. These programs are written in Idris, with no explicit erasure annotations, and are compiled to native code using the standard Idris pipeline.

Palindrome reads one line from the standard input as an Idris `String` and unpacks it into a list of characters. Then it checks whether the list is a palindrome, using the presented algorithm, and then prints `"yes"` or `"no"`.

In my benchmarks, the input string was always of the form `"abbb...bbba"` and the answer was thus always `"yes"`. This also means that short-circuiting `and` was not possible.

The input size is the length of the input string.

Binary adder reads four lines from the standard input as strings. The first line is parsed as a decadic natural number w, the width of the upcoming two binary numbers. Each of the following two lines is unpacked into a list of characters, reversed, and then the first w characters (either '0' or '1') are used to build a w-bit number, possibly padded with zeros up to the width w. The fourth number is a decadic natural number n giving how many times addition should be repeated.

The program converts the resulting sum after n iterations into an Idris `String` of length $w + 1$, which is then printed.

Strictly speaking, the conversion to string in the last step takes quadratic time because Idris strings have to be copied with every cons. However, the adder performs enough work to make this overhead negligible.

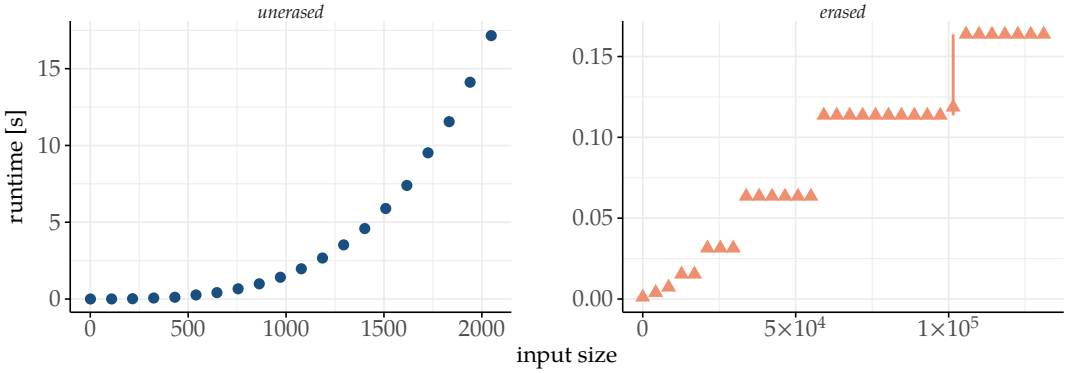
In my benchmarks, both input numbers were identical: '1' followed by $w - 1$ zeroes. The number of iterations n was always 10^5.

The input size is the number w.

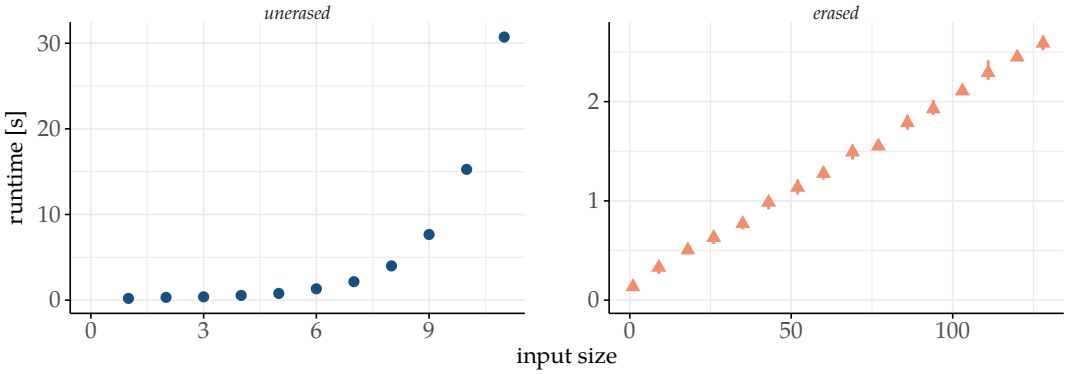
RLE reads a decadic natural number n from the standard input. Then it creates a list of n integers (`Int`) with each element equal to 1. This list is RLE-compressed and decompressed, and the length of the result is printed as a decadic natural number.

The input size is the number n.

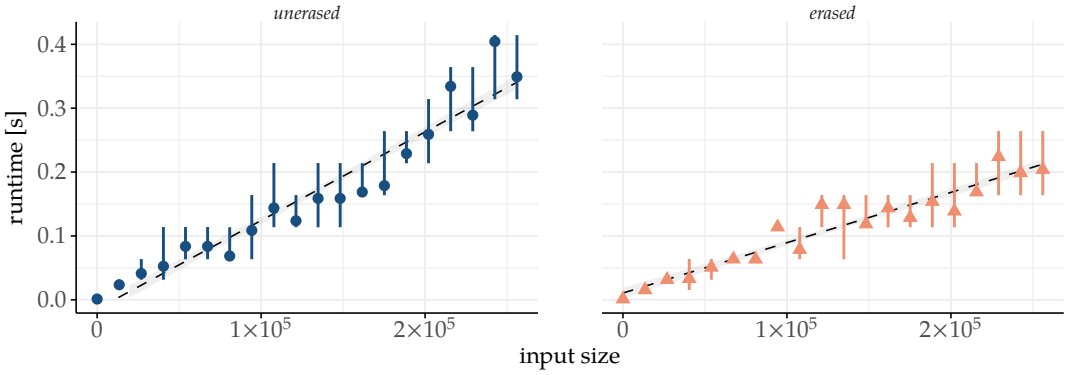
Idris special-cases (unary) natural numbers in the compiler and represents them as big GMP integers. However, non-primitive operations on naturals use exclusively their unary structure and therefore even if their size is logarithmic, operations on them may still be linear or worse.



(A) Palindrome checker



(B) Binary adder (10^5 iterations)



(C) RLE

FIGURE 4.8: Run times of erased and unerased programs
Points show the mean and whiskers extend to the minimal and maximal sample for each input size. Vertical axis shows runtime in seconds.
Note that each plot, except for the two RLE plots, has different *scales*, both vertically and horizontally.

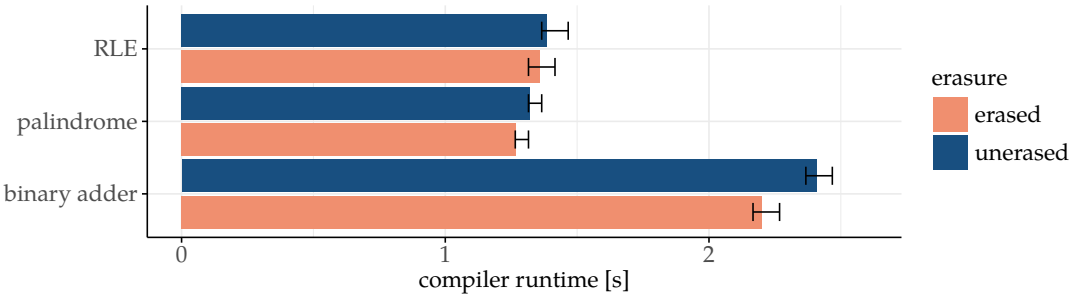


Figure 4.9: Compilation times of erased and unerased programs, including erasure analysis
Horizontal axis shows compilation time in seconds. Includes every stage, from parsing to code generation. Bars show mean, error bars extend to the minimal and maximal sample. This includes the whole run time of the Idris compiler, including parsing on one end and native code generation via C on the other.

4.6.1.2 Results

I ran each program, with and without erasure, on inputs of different size, 10 times per input, on a desktop computer. I do not report on the hardware configuration or other details since here we focus only on the asymptotic behaviour of the programs.

Since Idris 1 does not have a command line option to enable or disable erasure, the unerased programs were compiled by the last stable version of Idris without erasure (v0.9.12), and the erased programs were compiled by a version built some time after merging the erasure patch.

Palindrome Figure 4.8a shows the run time of a palindrome decider. In Section 3.1.2.2, we established that the indexing data must lead to a quadratic slowdown. Linear regression in the log-log plot shows that my implementation takes *cubic* time to compute the result (see Section 4.6.2).

However, the same program, compiled with an erasing compiler exhibits the behaviour seen on the left, where the run time seems to be dominated by effects of the Idris RTS, possibly garbage collection. In any case, linear regression in the log-log plot indicates that this program is likely linear.

Binary adder Figure 4.8b shows the run time of the binary adder. The unerased variant slows down exponentially, which is especially visible in the rightmost samples, which take approximately < 5, 7.5, 15 and 30 seconds.

The run time of the erased program grows only linearly.

Run-length encoder Finally, Figure 4.8c shows that erasure can help even if it does not improve asymptotic behaviour of a program.

Thanks to sharing (Figure 3.1), the in-memory size of the compressed list, as well as the run time of compression and decompression, is linear.

[4]These benchmarks, and others, can be found online at https://github.com/ziman/idris-benchmarks/.

Unlike the previous two cases, both plots in this figure use *the same* scale for the axes of both plots. This reveals that the run times of the unerased program rise slightly more steeply than those of the erased program.

Therefore, erasure does not yield asymptotic improvements, but eliminates unnecessary index manipulation.

Compilation time Figure 4.9 also shows that erased programs take slightly less time to compile. This may seem surprising at first, but programs with heavy use of dependent typing contain a lot of erasable code. While the erasure analysis does take some time, there is less code to generate by the later stages of the compiler.

The plot includes all stages of compilation, such as parsing, erasure inference, Scheme generation, compilation of Scheme to native code via C using Chicken Scheme [CHI20].

4.6.2 Comparison with theory

The expected and measured results are summarised in Table 4.10.

Assuming that the time complexity of all programs except for the unerased binary adder is $O(n^k)$, we can calculate an approximation of k for each program using linear regression in the log-log plot of its run time. We estimate k as the slope of the best linear fit, as further discussed in Section 9.1.4.

Before fitting, I removed all samples below a certain input size to reduce the effect of lower-order terms. For Palindrome, both erased and unerased, I removed all samples for inputs shorter than 1000 elements. For binary numbers (erased), I removed all samples for inputs shorter than 2 binary digits. For RLE, both erased and unerased, I removed all samples for inputs smaller than 1000 elements.

Table 4.10 shows that in all programs that we expect to be linear, which includes all erased programs and the unerased RLE program, we obtain exponent estimates within 1.0 ± 0.2.

My implementation of the palindrome checker seems to be *cubic* when not erased. This contradicts the expected complexity $O(n^2)$, which would have to be relaxed to $\Omega(n^2)$. I have not found the reason but I also have not investigated it deeply.

Finally, the unerased binary adder has no exponent measured because it does not satisfy the assumptions of the model (polynomiality). Indeed, even in the log-log plot, the run times of the unerased binary adder form a convex curve.

4.7 Discussion

I have shown that even such a simple erasure scheme as the one presented in this chapter can remove non-computational data from programs where existing systems fall short (as shown in Section 3.2) and recover the desired linear complexity in the example programs.

Program	Expected complexity		Measured exponent	
	erased	unerased	erased	unerased
Palindrome checker	$O(n)$	$O(n^2)$	1.11 ± 0.01	3.38 ± 0.01
Binary adder	$O(n)$	$O(2^n)$	0.81 ± 0.01	—
Run-length decoder	$O(n)$	$O(n)$	0.89 ± 0.03	0.96 ± 0.03

TABLE 4.10: Complexities of the example programs
Measured exponents are given ± one standard error.

Furthermore, experience with its implementation in Idris 1 over the past six years shows that especially with several extensions (Section 4.5), this method is suitable for use in practical programming languages.

Erasure of constructor fields Compared to most other approaches to useless variable elimination, the erasure approach in this chapter can erase constructor fields, not just arguments of functions. This is a necessary feature if we are designing erasure for dependently typed languages.

Non-invasiveness An interesting property of this erasure scheme is that it is easy to add to an existing language. It does not modify the core calculus; instead, it computes usage "on the side" and then produces a list of *nodes* saying which argument of which function is erasable.

In other words, the set of programs that are accepted by a programming language remains unchanged by adding this erasure mechanism.

Surprising effects of erasure inference Erasure inference can produce unexpected results. One of the most frequent effects is the occasional erasure of the recursive field in recursive data types, such as in the constructor S of the naturals, or the tail of a vector in the constructor (::) of vectors.

This happens for example if the main function computes a value of such a type and just returns it instead of printing it. Since the erasure analysis discovers that all inspections of the recursive occurences are used only to construct other recursive occurences, it is consistent to remove all of them.

Then the program computes only that the answer is a successor (returns just S) or computes only the head of the list but not the rest (returns (::) 42).

This resembles "static laziness": some computation is not only left unevaluated, it is removed entirely, at compile time.

Laziness This method is also applicable to lazy languages since lazy evaluation only delays usage; it does not add new usage. Thus an usage pattern inferred with strict evaluation in mind will be a safe approximation for lazy evaluation of the same program.

Dependence on case-tree elaboration Since the operational behaviour of a program depends on how patterns are compiled to case trees, erasure depends on it as well. Programmers need suitable tools to be explicit about the desired operational behaviour where necessary.

This would remedy the problem with double-purpose usage of erasure annotations in Idris 1 – both as erasure assertions, but also as a way to select forced patterns in a pattern match (see Section 4.5.1).

Untypedness Another interesting aspect of this erasure scheme is that it ignores types: the only purpose types are used for is determining the arity of definitions. This erasure scheme could thus also be useful for other untyped languages with data types and pattern matching – most likely an intermediate representation of another functional language.

4.7.1 Shortcomings

Higher-order functions As mentioned earlier, this erasure scheme needs extensions to erase from functional arguments of functions and data constructors, let expressions or lambdas. The extension for the one of the most common instances of this problem, type classes, has been implemented in Idris 1 (Section 4.5.3) and it works well.

However, the more general case needs more general extensions and some problems do not seem to be solvable at all in the current scheme, such as functions with computed types (Section 4.5.5).

Placeholders instead of argument removal In data constructors, this erasure method *removes* fields entirely, which means that at runtime, a constructor has a different arity. For functions, this method merely fills the unused arguments with the undefined term □ in each application.

There is no fundamental reason for not removing arguments from functions at runtime, except that careful design and implementation is needed, especially in the questions of partial application and higher-order variables.

Whole-program analysis The negation-as-failure approach to erasure inference means that even if we can analyse separate functions separately, we generally cannot conclude that a value is erasable until we have seen the whole program – each piece of the program might theoretically produce a constraint that transitively causes the value in question to be needed at runtime.

This "action at distance" happens through arguments of constructors and Section 7.4 discusses how we can make erasure inference more modular.

Token type target elimination The problem with erasure of token type target elimination (discussed in Section 9.2.1.8), where the reduction behaviour (especially strong normalisation) of a calculus is not preserved after erasure, does *not* apply to

this erasure method because we do not change arities of functions. Instead, we pass □ in place of all erased arguments (Section 4.4), which prevents the reduction problems.

Non-argument nodes It would be useful to place erasure nodes not only on arguments on functions but also on each binder, each let-bound definition, on each application of two terms, on each scrutinee of a case expression, etc. – this is the approach taken in Chapter 5.

This would solve problems such as the slightly awkward erasure rule for single-branch case trees (Figure 4.6):

$$\left\langle \begin{array}{l} \textbf{case } v \textbf{ of} \\ \quad \mathsf{n}\ v_1\ v_2\ \ldots\ v_m\ \Rightarrow\ B \\ \textbf{if} \\ \quad \forall i.\ v_i \text{ not free in } \langle B \rangle_S \end{array} \right\rangle_S = \langle B \rangle_S$$

This rule says that we must check whether any of the variables v_i is free in the erased RHS, $\langle B \rangle_S$, in order to figure out whether we can eliminate the case inspection as useless.

If we had an erasure node x attached to the case expression, whose usage would be implied by the usage of any of $v_1 \ldots v_m$, we could decide whether to elide the case inspection immediately by checking whether $x \in S$, instead of traversing $\langle B \rangle_S$ in search of any occurrence of $v_1 \ldots v_m$.

The same approach would help erase let-bound definitions.

Non-covering definitions The erasure transformation in this chapter does not preserve pattern match failures in non-covering programs. Consider the following program.

```
isZero : ℕ → Bool
isZero Z = True
— missing clause for S n

main : Bool
main = isZero (S Z)
```

After erasure, this program reduces to True instead of failing with a pattern match failure. We have erased too much.

On the other hand, Idris allows non-covering function definitions but non-covering pattern matches are compiled into trees with extra default cases that abort the program with an error message at runtime. These extra cases cause inspection of all arguments of non-covering functions (and also constructors in nested matches), which prevents their erasure and also transitively blocks erasure in other parts of the program. This means that we now erase *too little*.

To improve this situation, we could:

- Add the extra inspecting cases *after* erasure analysis. If the pattern match fails, the program will abort immediately and it does not matter that the right-hand side of the function is not erasure-consistent.
- Devise a more careful way of adding the extra cases only where absolutely necessary instead of in every case inspection, which might preserve erasure better.
- Use a more elaborate erasure approach, such as the one in Chapter 5.

It remains to be seen which approach is the most useful.

Relationship to typed erasure Chapters 5 and 6 describe a type-based approach to erasure. It is an open problem how the approach in this chapter formally relates to the type-based approach. Any description of a relationship will require a development of suitable metatheory for IR, which I do not provide in this dissertation.

Chapter 5

A dependent calculus with erasure

A way to eliminate the drawbacks of the erasure approach from Chapter 4, is using a core calculus that supports erasure natively.

This chapter presents a new dependently typed calculus, which has erasure built into its syntax and typing rules. The name of the calculus is $\mathsf{TT}_\star$ and it has been designed as an extension of TT, the core calculus of Idris [Bra13]. However, its principles should be applicable to other calculi, underlying other dependently typed languages.

I give a description of the calculus, its reduction and typing rules, show how erasure works with it, and prove several desirable properties of it, such as confluence, subject reduction, and soundness of erasure. Inference of erasure annotations is described separately in Chapter 6.

5.1 Introduction

Figure 5.1 shows the compilation pipeline of Idris. $\mathsf{TT}_\star$ is a *core calculus*, which means that it is fully explicit and contains a lot of type/erasure annotations.

As explained in Section 2.1.6, this verbosity makes it easy to check automatically but also tedious to write by hand. Implementations are expected to have a *surface language*, which allows omission of machine-inferrable details (such as type and erasure annotations) in the input provided by the programmer, and an *elaborator*, which fills in all the blanks to obtain a fully annotated program expressed in the core calculus.

This dissertation discusses elaboration of erasure annotations in Chapter 6. However, I do not discuss elaboration of *type* annotations, which I assume to be done using established methods [Bra13].

$\mathsf{TT}_\star$ inherits mostly from TT of Idris (core calculus) and EPTS of Mishra-Linger [ML08] (erasure semantics). The most important difference between TT and $\mathsf{TT}_\star$ is that the syntax of $\mathsf{TT}_\star$ includes explicit *erasure annotations* on all name binders and applications. There are also other differences, such $\mathsf{TT}_\star$ **let**-binding arbitrary pattern matching functions and all type and data constructors, instead of

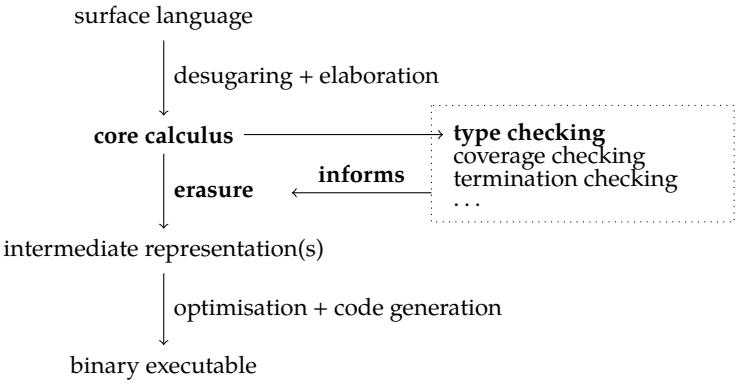


FIGURE 5.1: High-level overview of a typical compilation process, as found in Idris

defining them in a top-level, global scope. The most important difference from Mishra-Linger's EPTS [ML08] is support of full dependent pattern matching, including forced patterns and forced constructors.

5.2 Syntax

The syntax of $\mathsf{TT}_\star$ is shown in Figure 5.2. The calculus has variables, lambda expressions, dependent type expressions, let expressions, and applications. The special term $\square$, "omitted", replaces types after erasure (everything to the right of colons). There is no special syntax for the type of types; Type is just a name.

5.2.1 Erasure annotations

The key aspect of $\mathsf{TT}_\star$ is that every binder (colon) and every application is annotated with an erasure annotation, which can have the values shown in Table 5.4 – like in Mishra-Linger's EPTS [ML08].

In some sentences, it is more natural to use the expression "retention annotations" instead of "erasure annotations" since the annotations behave like Boolean values if True is interpreted as "retain (do not erase)".

Annotation	Meaning
$\bullet$	Erasability unknown/missing.
i	Erasure variable number i; not a specific value.
R	Runtime/computational value. Retain. Not erasable.
E	Erasable.

TABLE 5.4: Erasure annotations in $\mathsf{TT}_\star$

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

It turns out to be useful to create several different "flavours" of $\mathsf{TT}_\star$ for different stages of the erasure pipeline (Figure 5.5), restricting which erasure annotations are

In this figure, n stands for names, i stands for integers, and $\overline{X}$ stands for zero or more repetitions of X.

term	::=	n	— *reference to name*
		$\mid \lambda n :_r term.\ term$	— *lambda*
		$\mid (n :_r term) \to term$	— *dependent product*
		$\mid$ **let** *def* **in** *term*	— *let binding*
		$\mid term \mathbin{\widehat{r}} term$	— *application*
		$\mid \square$	— *omitted type*
r	::=	$\mathsf{R} \mid \mathsf{E} \mid i \mid \bullet$	— *erasability*
env	::=	$\overline{def}$	— *environment*
def	::=	$n :_r term = body$	— *definition*
body	::=	**constructor** $\mid$ **variable**	— *abstract definition*
		$\mid term \mid \overline{clause}$	— *concrete definition*
clause	::=	$\overline{(n :_r term)}.\ pat = term$	— *pattern clause*
pat	::=	n	— *reference to name*
		$\mid pat \mathbin{\widehat{r}} pat$	— *application*
		$\mid [term]$	— *forced pattern*
		$\mid \lceil c \rceil$	— *forced constructor*
		$\mid \lfloor f \rfloor$	— *name of whole definition*

FIGURE 5.2: Syntax of $\mathsf{TT}_\star$

T, M, N, R, F, G, X, Y	term
$\alpha, \beta, \tau, \sigma, \rho, \nu$	type (i.e. term)
$\Delta, \Sigma, \Phi, \Xi, \Psi, \Lambda, P$	constraint set
n, m, f, x, c	name
r, s, t, u, v, q	erasure annotation

Γ, Π	environment	b	definition body
μ, π	substitution	P, L	pattern
d, p	definition	$\overline{x}, \overline{x}^{\,i}, \overline{x}^{\,1..n}$	sequence $\{x_i\}_{1 \le i \le n}$
C	clause	$\emptyset$	empty sequence
i, j	evar number		

FIGURE 5.3: Notation conventions

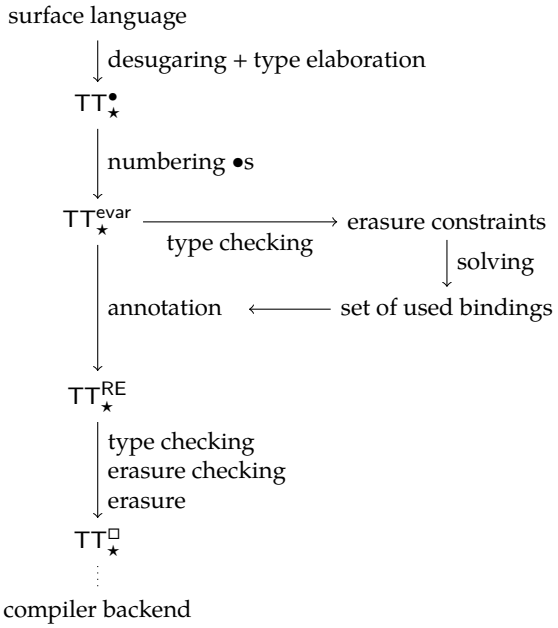


FIGURE 5.5: Inference, checking and erasure process of $\mathsf{TT}_{\star}$

allowed at which stage. This helps with presentation of the pipeline on paper, but also in the implementation, where I parameterise the inductive type of $\mathsf{TT}_{\star}$ terms with the type of erasure annotations to obtain the different variants as separate types.

Variant	Definition of r	Purpose
$\mathsf{TT}^{\bullet}_{\star}$	$r ::= \mathsf{R} \mid \mathsf{E} \mid \bullet$	optional annotations given by programmer
$\mathsf{TT}^{\mathsf{evar}}_{\star}$	$r ::= \mathsf{R} \mid \mathsf{E} \mid i$	*evars* numbered for erasure inference
$\mathsf{TT}^{\mathsf{RE}}_{\star}$	$r ::= \mathsf{R} \mid \mathsf{E}$	final form, fully explicit, checkable
$\mathsf{TT}^{\square}_{\star}$	$r ::= \bullet$	output of erasure

TABLE 5.6: Variants of $\mathsf{TT}_{\star}$

The variants are defined by varying the production rule for r, shown in Table 5.6.

$\mathsf{TT}^{\bullet}_{\star}$ is used for the output of (type) elaboration, where erasure annotations remain as given by the programmer – either explicit or left for the computer to figure out.

In $\mathsf{TT}^{\mathsf{evar}}_{\star}$, all undefined annotations are replaced with erasure variables, shortly *evars*. Evars are numbered and their purpose is to represent erasure annotations found in the program. They appear in erasure constraints in order to express relationships between erasure annotations. (For further details, see Section 6.1.1.)

$\mathsf{TT}^{\mathsf{RE}}_{\star}$ is a fully explicit, fully annotated, central core calculus, which contains only definite annotations (either R or E), and it is this calculus that can be checked using the typing rules given in Section 5.5 to obtain the correctness guarantees given in Section 5.7.

Finally, $\mathsf{TT}^{\square}_{\star}$ is used to express erased programs that do not contain any erasure annotations anymore. Because erasure removes terms to the right of colons, too, these

programs are essentially untyped, too[1].

5.2.2 Definitions in $\mathsf{TT}_\star$

Let expressions and environments bind *definitions*, which give the type, erasability, and body for the name being defined. In this dissertation and its implementation, I represent *all* binders as definitions, including lambdas and Π expressions. This comes with the following notation conventions.

- If the erasure annotation is $\bullet$ (unknown), we can omit it from the colon subscript.

  $$(n : \tau = b) := (n :_\bullet \tau = b)$$

- If the erasure annotation is $\bullet$ (unknown) and the type is $\Box$ (omitted), we can omit the colon, too.

  $$(n = b) := (n :_\bullet \Box = b)$$

- If the body of the definition is **variable**, we can omit it.

  $$(n :_r \tau) := (n :_r \tau = \textbf{variable})$$

The above rules allow us to write $(\lambda x.\ x)$ for $(\lambda x :_\bullet \Box = \textbf{variable}.\ x)$, which is useful especially for terms in $\mathsf{TT}^\Box_\star$.

5.2.2.1 Definition bodies

Definitions in $\mathsf{TT}_\star$ can have four different kinds of bodies:

term T, where the name being defined stands for the term T;

clauses $\overline{C}$, where the name being defined stands for the function described by the sequence of pattern matching clauses $\overline{C}$;

constructor, where the name being defined stands for itself; most importantly, it never reduces to anything else than itself;

variable, where the name being defined stands for some other (possibly unknown) value and may reduce to something else than itself.

The difference between constructors and variables shows in reduction of pattern clauses (Section 5.4.3).

[1] It is possible to have a typed calculus for the results of erasure – Mishra-Linger uses a calculus called IPTS [ML08] based on Miquel's ICC [Miq01] for that purpose – but these aspects of the post-erasure representation are out of the scope of this dissertation.

5.2.2.2 Inductive type families

$\mathsf{TT}_\star$ does not have any special facility for defining inductive families and an inductive family is defined using let bindings with **constructor** bodies, one for its type constructor, plus one for each data constructor.

This does not preserve the connection between data constructors and their families (their type constructors) but $\mathsf{TT}_\star$ does not need this information anywhere. Ensuring that data constructors have types appropriate for their families, coverage checking, and related tasks is therefore a responsibility of the elaborator of any particular implementation.

However, if needed, $\mathsf{TT}_\star$ can be easily extended to support grouping of definitions into type families.

Example Let us elaborate the standard definition of length-indexed vectors into the core form in $\mathsf{TT}_\star$.

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Vect} : (n : \mathbb{N}) \to (a : \mathsf{Type}) \to \mathsf{Type}\ \textbf{where} \\
\quad \mathsf{Nil}\ :\ \mathsf{Vect}\ \mathsf{Z}\ a \\
\quad (::)\ :\ (x : a) \to (xs : \mathsf{Vect}\ n\ a) \to \mathsf{Vect}\ (\mathsf{S}\ n)\ a
\end{array}
$$

The above definition in the surface language is elaborated into $\mathsf{TT}_\star$ as follows.

$$
\begin{array}{l}
\textbf{let}\ \mathsf{Vect} : (n : \mathbb{N}) \to (a : \mathsf{Type}) \to \mathsf{Type} = \textbf{constructor}\ \textbf{in} \\
\quad \textbf{let}\ \mathsf{Nil} : (a : \mathsf{Type}) \to \mathsf{Vect}\ \mathsf{Z}\ a = \textbf{constructor}\ \textbf{in} \\
\quad\quad \textbf{let}\ (::) : (a : \mathsf{Type}) \to (n : \mathbb{N}) \to (x : a) \to (xs : \mathsf{Vect}\ n\ a) \to \mathsf{Vect}\ (\mathsf{S}\ n)\ a \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad = \textbf{constructor} \\
\quad\quad \textbf{in}\ \ldots
\end{array}
$$

Being a core calculus, $\mathsf{TT}_\star$ does not feature implicit parameters so instead of e.g. $\{a : \mathsf{Type}\}$, we wrote $(a : \mathsf{Type})$ above.

5.2.3 Pattern matching clauses

To keep the presentation straightforward, $\mathsf{TT}_\star$ features definitions by pattern matching clauses as its pattern matching facility. I expect that implementations may want to use case trees in the core language instead, which is discussed in Section 7.2, where I describe type checking of case trees by conversion to pattern clauses.

Each pattern matching clause contains explicit definitions (type annotations) of pattern variables, the pattern on the LHS, and the term on the RHS.

Pattern variables are variables bound on the LHS of a pattern clause and they stand for values arising from pattern matching. Like all binders, they have to be given explicit types. The grammar of patterns contains:

names, which in well-typed patterns refer to either constructors or pattern variables;

applications of a pattern to another pattern;

forced patterns and forced constructors, which express that if the rest of the pattern matches, then from well-typedness of the program, we know that the corresponding value will always be the same as the term in the square brackets.

Forced patterns are introduced in Section 2.1.4 and characterised in Definition 5.13 below.

definition names, which name the whole function being defined by the pattern clause. In well-formed left-hand sides of pattern clauses, they appear exactly once: at the head of the left-hand side pattern (application).

5.2.4 Environments

Typing/evaluation environments are telescopes of definitions. In general, we will consider any sequence of definitions *telescopic*, which means that the types and bodies of definitions bound later in the sequence can refer to the definitions bound earlier in the sequence, but not vice versa.

5.3 Assumptions

To focus on the presentation of erasure, I assume that implementations will implicitly take care of the following points.

Name clashes We will assume that all binders in any program bind distinct names. This is to avoid problems with name clashes, which are difficult to address formally while retaining the clarity of explanation, but also entirely uninteresting and irrelevant to the topic of erasure. Disjointness of binders also means that we assume that all names bound in a well-formed environment are distinct from each other and from all other names occurring in the given program. This is similar to the *Barendregt convention* [Pie02, Chap. 6].

In other words, we will ignore the issue of name clashes, assuming that they are taken care of either by a suitable representation of terms, by a suitable variable-renaming pass before processing the program, by a careful implementation of the calculus, or by any other means.

Termination This dissertation does not discuss termination of $\mathsf{TT}_\star$ programs. Instead, it assumes that implementations will provide their own termination checkers in a standard way [LJBA01; Abe98].

Pattern coverage This dissertation does not discuss nor assume pattern coverage. Again, practical implementations are expected to provide their own coverage checkers.

The main correctness result, that erasure commutes with reduction, holds trivially if no reduction occurs due to incomplete pattern coverage.

Forced patterns I expect the implementations to also check the consistency of forced patterns, as characterised in Definition 5.13.

Type-in-Type $\mathsf{TT}_\star$ uses type-in-type to make the presentation clearer, since this question is orthogonal to erasure. Implementations are expected to bring their own universe stratification.

Typecase $\mathsf{TT}_\star$ has no means to inspect Π expressions. We therefore always erase both sides of function arrows. Once future work establishes a way to pattern match on Π, the reduction rules and the typing rules will have to be adjusted accordingly.

5.4 Reduction rules

The reduction rules of $\mathsf{TT}_\star$ are split into several groups. Figure 5.10 gives the computation rules, while Figures 5.11, 5.12 and 5.13 provide its structural closure.

$\mathsf{TT}_\star$ does not prescribe any evaluation strategy.

Definition 5.1 (Term substitution)**.** We write $T[x \mapsto X]$ for substitution of term X in place of all (free) occurrences of variable x in term T.

5.4.1 Reduction in terms

Rule RedLetElim allows removal of an unreferenced let expression, and rule RedLetAppL allows floating applications into let expressions. This is necessary for let expressions that bind pattern matching functions.

Rule RedVar replaces references with their definitions, and Redex provides β-reduction.

To state and explain RedClauses, we need to introduce further definitions.

5.4.2 Reduction with pattern clauses: definitions

Definition 5.2 (Well-formed pattern)**.** We write $\Gamma; \Pi \vdash \textsc{PatWF}_f(L)$ to express that L is a well-formed LHS of a pattern clause of function f in environment Γ with pattern variables Π.

This judgement is defined in Figure 5.7 and it checks only that the names of the correct kind – pattern variables, constructors, function names – occur in correct places. For example, well-formed patterns don't contain applications of pattern variables.

We write PatWF for "any of $\textsc{PatWF}_f$ for some f, $\textsc{PatWF}_F$, or $\textsc{PatWF}_X$".

Definition 5.3 (Substitution functions)**.** A *substitution function* (or, shortly, *substitution*) μ is a function from names to terms.

As a special case, we write "$x \mapsto X$" for the function that maps name x to term X and any other name y to term y (i.e. variable with name y).

For any substitution μ, we write $T[\mu]$ to denote substitution of μ into term (or pattern) T. This generalises Definition 5.1.

predicate	used for	allowed pattern forms
$\textsc{PatWF}_f$	LHS of clause	applications, $\lfloor f \rfloor$
$\textsc{PatWF}_F$	LHS of application	applications, constructors, forced constructors
$\textsc{PatWF}_X$	RHS of application	any
PatWF	any of the above	

$$\frac{\Gamma;\Pi \vdash \textsc{PatWF}_f(F) \qquad \Gamma;\Pi \vdash \textsc{PatWF}_X(X)}{\Gamma;\Pi \vdash \textsc{PatWF}_f(F \mathbin{\widehat{r}} X)}\ \textsc{PatWF}_f\text{-App} \qquad \frac{(f :_s \sigma = \overline{C}) \in \Gamma}{\Gamma;\Pi \vdash \textsc{PatWF}_f(\lfloor f \rfloor)}\ \textsc{PatWF}_f\text{-Head}$$

$$\frac{\Gamma;\Pi \vdash \textsc{PatWF}_F(F) \qquad \Gamma;\Pi \vdash \textsc{PatWF}_X(X)}{\Gamma;\Pi \vdash \textsc{PatWF}_F(F \mathbin{\widehat{r}} X)}\ \textsc{PatWF}_F\text{-App}$$

$$\frac{(c :_s \sigma = \textbf{constructor}) \in \Gamma}{\Gamma;\Pi \vdash \textsc{PatWF}_F(c)}\ \textsc{PatWF}_F\text{-Ctor} \qquad \frac{(c :_s \sigma = \textbf{constructor}) \in \Gamma}{\Gamma;\Pi \vdash \textsc{PatWF}_F(\lceil c \rceil)}\ \textsc{PatWF}_F\text{-ForcedCtor}$$

$$\frac{\Gamma;\Pi \vdash \textsc{PatWF}_F(F) \qquad \Gamma;\Pi \vdash \textsc{PatWF}_X(X)}{\Gamma;\Pi \vdash \textsc{PatWF}_X(F \mathbin{\widehat{r}} X)}\ \textsc{PatWF}_X\text{-App}$$

$$\frac{(c :_s \sigma = \textbf{constructor}) \in \Gamma}{\Gamma;\Pi \vdash \textsc{PatWF}_X(c)}\ \textsc{PatWF}_X\text{-Ctor} \qquad \frac{(c :_s \sigma = \textbf{constructor}) \in \Gamma}{\Gamma;\Pi \vdash \textsc{PatWF}_X(\lceil c \rceil)}\ \textsc{PatWF}_X\text{-ForcedCtor}$$

$$\frac{T \text{ is a term}}{\Gamma;\Pi \vdash \textsc{PatWF}_X([T])}\ \textsc{PatWF}_X\text{-Forced} \qquad \frac{(n :_s \sigma) \in \Pi}{\Gamma;\Pi \vdash \textsc{PatWF}_X(n)}\ \textsc{PatWF}_X\text{-Patvar}$$

$$\frac{\Gamma;\Pi \vdash \textsc{PatWF}_f(P) \text{ for some } f}{\Gamma;\Pi \vdash \textsc{PatWF}(P)}\ \textsc{PatWF}\text{-}f \qquad \frac{\Gamma;\Pi \vdash \textsc{PatWF}_F(P)}{\Gamma;\Pi \vdash \textsc{PatWF}(P)}\ \textsc{PatWF-F} \qquad \frac{\Gamma;\Pi \vdash \textsc{PatWF}_X(P)}{\Gamma;\Pi \vdash \textsc{PatWF}(P)}\ \textsc{PatWF-X}$$

Figure 5.7: Well-formed patterns

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

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

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

Figure 5.8: Pattern match
The general form of the judgement is $\Gamma;\Pi \vdash P \parallel_\mu T$ for pattern P, term T, and substitution μ.

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

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

Figure 5.9: 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 sequence of arguments $\overline{X}$ may be empty.

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

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

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

Figure 5.10: Reduction: computation rules

$$\frac{\Gamma \vdash F \rightsquigarrow F'}{\Gamma \vdash F \mathrel{\widehat{r}} X \rightsquigarrow F' \mathrel{\widehat{r}} X}\ \textsc{RedAppL} \qquad \frac{\Gamma \vdash X \rightsquigarrow X'}{\Gamma \vdash F \mathrel{\widehat{r}} X \rightsquigarrow F \mathrel{\widehat{r}} X'}\ \textsc{RedAppR}$$

$$\frac{\Gamma \vdash \sigma \rightsquigarrow \sigma'}{\Gamma \vdash \lambda n :_s \sigma.\, T \rightsquigarrow \lambda n :_s \sigma'.\, T}\ \textsc{RedLamL} \qquad \frac{\Gamma, (n :_s \sigma) \vdash T \rightsquigarrow T'}{\Gamma \vdash \lambda n :_s \sigma.\, T \rightsquigarrow \lambda n :_s \sigma.\, T'}\ \textsc{RedLamR}$$

$$\frac{\Gamma \vdash \sigma \rightsquigarrow \sigma'}{\Gamma \vdash (n :_s \sigma) \to \rho \rightsquigarrow (n :_s \sigma') \to \rho}\ \textsc{RedPiL} \qquad \frac{\Gamma, (n :_s \sigma) \vdash \rho \rightsquigarrow \rho'}{\Gamma \vdash (n :_s \sigma) \to \rho \rightsquigarrow (n :_s \sigma) \to \rho'}\ \textsc{RedPiR}$$

$$\frac{\Gamma \vdash d \rightsquigarrow d'}{\Gamma \vdash \textbf{let}\ d\ \textbf{in}\ T \rightsquigarrow \textbf{let}\ d'\ \textbf{in}\ T}\ \textsc{RedLetL} \qquad \frac{\Gamma, d \vdash T \rightsquigarrow T'}{\Gamma \vdash \textbf{let}\ d\ \textbf{in}\ T \rightsquigarrow \textbf{let}\ d\ \textbf{in}\ T'}\ \textsc{RedLetR}$$

Figure 5.11: Reduction: structural rules for terms

$$\frac{\Gamma \vdash \sigma \rightsquigarrow \sigma'}{\Gamma \vdash (n :_s \sigma = b) \rightsquigarrow (n :_s \sigma' = b)}\ \textsc{RedDefType} \qquad \frac{\Gamma, (n :_s \sigma = T) \vdash T \rightsquigarrow T'}{\Gamma \vdash (n :_s \sigma = T) \rightsquigarrow (n :_s \sigma = T')}\ \textsc{RedDefTerm}$$

$$\frac{\Gamma, (n :_s \sigma = \overline{C}) \vdash \overline{C} \rightsquigarrow \overline{C'}}{\Gamma \vdash (n :_s \sigma = \overline{C}) \rightsquigarrow (n :_s \sigma = \overline{C'})}\ \textsc{RedDefClauses} \qquad \frac{\Gamma \vdash \Pi \rightsquigarrow \Pi'}{\Gamma \vdash (\Pi.\ L = R) \rightsquigarrow (\Pi'.\ L = R)}\ \textsc{RedClausePi}$$

$$\frac{\Gamma; \Pi \vdash L \rightsquigarrow L'}{\Gamma \vdash (\Pi.\ L = R) \rightsquigarrow (\Pi.\ L' = R)}\ \textsc{RedClauseL} \qquad \frac{\Gamma, \Pi \vdash R \rightsquigarrow R'}{\Gamma \vdash (\Pi.\ L = R) \rightsquigarrow (\Pi.\ L = R')}\ \textsc{RedClauseR}$$

$$\frac{\Gamma, \Pi \vdash d \rightsquigarrow d'}{\Gamma \vdash (\Pi, d) \rightsquigarrow (\Pi, d')}\ \textsc{RedTeleHere} \qquad \frac{\Gamma \vdash \Pi \rightsquigarrow \Pi'}{\Gamma \vdash (\Pi, d) \rightsquigarrow (\Pi', d)}\ \textsc{RedTeleThere}$$

Figure 5.12: Reduction: structural rules for non-terms

$$\frac{\Gamma, \Pi \vdash T \rightsquigarrow T'}{\Gamma; \Pi \vdash [T] \rightsquigarrow [T']}\ \textsc{RedPatForced}$$

$$\frac{\Gamma; \Pi \vdash F \rightsquigarrow F'}{\Gamma; \Pi \vdash (F \mathrel{\widehat{s}} X) \rightsquigarrow (F' \mathrel{\widehat{s}} X)}\ \textsc{RedPatAppL} \qquad \frac{\Gamma; \Pi \vdash X \rightsquigarrow X'}{\Gamma; \Pi \vdash (F \mathrel{\widehat{s}} X) \rightsquigarrow (F \mathrel{\widehat{s}} X')}\ \textsc{RedPatAppR}$$

Figure 5.13: Reduction: structural rules for patterns

Definition 5.4 (Bound variables)**.** We write $\mathrm{BV}(\Pi)$ for the set of names bound in environment Π.

$$n \in \mathrm{BV}(\Pi) \quad \Leftrightarrow \quad (n :_r \tau = b) \in \Pi \text{ for some } r, \tau, \text{ and } b$$

Definition 5.5 (Domain of substitution)**.** The domain of a substitution μ is the set of names that the substitution does not preserve.

$$\textsc{Dom}(\mu) := \{n \mid \mu(n) \neq n\}$$

Definition 5.6 (Pattern match)**.** Pattern P matches term T with substitution μ in contexts Γ with pattern variables Π if $\Gamma; \Pi \vdash P \parallel_\mu T$, as defined in Figure 5.8.

Remark 5.1. Rule MatchForcedCtor requires that the matched term is actually a constructor, even though we don't care *which* constructor it is. This is because otherwise we would obtain $\Gamma; \Pi \vdash \lceil \mathsf{S} \rceil\ n \parallel_\mu (+2)\ 3$ with $\mu(n) = 3$, which is undesirable.

Definition 5.7 (Pattern mismatch)**.** Pattern P mismatches term T in context Γ if $\Gamma \vdash P \nparallel T$, as given in Figure 5.9.

Intuitively, a pattern mismatch can arise only as a mismatch of constructors at some corresponding places in the pattern and term.

5.4.3 Reduction with pattern clauses

Rule RedClauses describes how pattern matching functions reduce. If the name f refers to a function given by pattern matching clauses $\overline{C}$, and the left hand side L_k of some clause C_k matches the term $f\ \overline{\hat{\tau} X}$, then this term reduces to the right hand side of C_k, subject to the appropriate substitution. Finally, all left-hand sides L_i must be well-formed so that the match or mismatch judgements make sense.

In order to perform this reduction step, we must be sure that all clauses preceding C_k have no chance of matching $f\ \overline{\hat{\tau} X}$, even after further reduction in $f\ \overline{\hat{\tau} X}$. This is approximated by the mismatch judgement $\Gamma \vdash L_i \nparallel f\ \overline{\hat{\tau} X}$, as proven in Lemmas 5.23 and 5.25.

Since in the mismatch judgement, the environment is used only to decide which names are constructors, it is sufficient to use Γ instead of Γ, Π.

5.4.3.1 Computing pattern matching

The pattern matching procedure takes environments $\Gamma; \Pi$, pattern P, and term T, and it can have three outcomes:

- Match μ – if $\Gamma; \Pi \vdash P \parallel_\mu T$;
- Mismatch – if $\Gamma; \Pi \vdash P \nparallel T$;
- Stuck – neither can be proven.

Upon encountering uncertainty that is not efficiently resolvable, we err on the side of Stuck, which blocks reduction of pattern matching until other forms of reduction bring the term to an obviously (mis-)matchable form.

The most important case of stuck reduction is matching a complex pattern against a term variable, which highlights the fundamental difference between variables and constructors – a constructor (on the term side) would be guaranteed not to reduce further and we could conclude such cases with Mismatch.

Figure 5.9 shows the pattern mismatching rules. There are subtle details in the rules.

- Rule MismatchHead allows a different number of arguments in the pattern than in the term. The other three rules require both applications to have the same depth.
- Each rule disregards erasure annotations, using r on the LHS and r' on the RHS. This means that erasure annotations cannot cause a pattern to mismatch.
- The depth of either application can be zero. This corresponds to mismatch of nullary constructors with no arguments.

5.5 Type- and erasure checking rules

As described in Section 5.2.1, the variant of $\mathsf{TT}_\star$ that is type- and erasure checked is $\mathsf{TT}^{\mathsf{RE}}_\star$, where the program is fully annotated and each erasure annotation is either a definite R or a definite E.

The rules given in this section are formulated as *type synthesis* rules, since type checking in $\mathsf{TT}_\star$ is performed by synthesising a type for any given term and then checking conversion of the synthesised type with the declared type.

All rules therefore take the environment, the context relevance and the term as the input, and compute the type of the term as the output.

5.5.1 Notation

For the typing rules, we will need to introduce more notation.

Remark 5.2 (Contexts and environments). In this dissertation, I use the word *environment* for typing or evaluation environments, such as Γ. I use the word *context* for the notion of how erasable the surroundings of a term are, e.g. "term appears in an erased context", "we check T in context r".

Definition 5.8 (Free pattern variables)**.** We write $\mathrm{FPV}_\Pi(P)$ for the set of pattern variables free in pattern P, where Π is the environment of pattern variables (Figure 5.18).

$$\begin{array}{c|cc} \wedge & \mathsf{E} & \mathsf{R} \\ \hline \mathsf{E} & \mathsf{E} & \mathsf{E} \\ \mathsf{R} & \mathsf{E} & \mathsf{R} \end{array} \qquad \frac{r \in \{\mathsf{E}, \mathsf{R}\}}{\mathsf{E} \leqslant r}\ \textsc{Impl-E} \qquad \frac{}{\mathsf{R} \leqslant \mathsf{R}}\ \textsc{Impl-R}$$

Figure 5.14: The erasability meet-semilattice

Definition 5.9 (Patterns-to-terms conversion)**.** We can convert pattern P to term $|P|$ as follows.

$$\begin{aligned} |n| &= n && \text{names (constructors and pattern variables)} \\ |F \mathrel{\widehat{r}} X| &= |F| \mathrel{\widehat{r}} |X| && \text{applications} \\ |\lceil T \rceil| &= T && \text{forced patterns} \\ |\lfloor f \rfloor| &= f && \text{definition names} \\ |\lceil c \rceil| &= c && \text{forced constructors} \end{aligned}$$

Definition 5.10 (Concatenation of environments)**.** For environments Γ and Γ', we write Γ, Γ' to express *concatenation* of Γ and Γ' as sequences.

Definition 5.11 (Pairs of environments)**.** We write $\Gamma; \Gamma'$ for the pair of environments Γ and Γ'.

Remark 5.3 (Environment concatenation vs. pairs). We need to define environment pairs for pattern typing rules, where we need to maintain the distinction between the (innermost) pattern variables and the outer environment. Intuitively, environment pairs should be seen almost as concatenations, except that we keep track of the boundary between the two constituents.

- Γ, d – extending environment Γ with definition d
- Γ, Π – extending environment Γ with sequence of definitions Π
- $\Gamma; \Pi$ – extending environment Γ with sequence of definitions Π, keeping track of the boundary

An example of usage of $\Gamma; \Pi$ can be see in the pattern typing rules in Figure 5.20.

5.5.2 Typing rules

Figure 5.14 defines the operator $\wedge$ and the relation $\leqslant$ for erasure annotations $\{\mathsf{E}, \mathsf{R}\}$. We will use these to define the typing rules.

5.5.2.1 Terms

The typing rules for $\mathsf{TT}^{\mathsf{RE}}_{\star}$ terms are shown in Figure 5.15.

Axiom I use type-in-type for simplicity of presentation. Implementations are welcome to bring their own universe stratification.

The typing judgement is available in any context r. If we wanted to enforce erasure of types, we could change the judgement to $\mathsf{Type} :_{\mathsf{E}} \mathsf{Type}$, which would enforce that types are used only in erasable contexts.

Ref This rule states that anytime a name is referred to, besides type-correctness, the name must be retained at least as much as the context in which the reference appears.

Pi Since $\mathsf{TT}_\star$ does not have any means to inspect the subterms of Π, we always consider them erased for simplicity.

Lam It is sufficient to check σ in an erased context because we know that σ will never be inspected at runtime. However, the term T needs to be checked with retention r.

In both Pi and Lam, the context is extended with $n :_s \sigma$, while McBride [McB16] would extend it with $n :_{s \wedge r} \sigma$. This difference is reconciled in my different formulation of Substitution Lemma, as illustrated in Section 8.4.1.1.

App The argument X is checked with retention $r \wedge s$, which means that X is used only if the whole application is used ($r = \mathsf{R}$) *and* function F uses its argument ($s = \mathsf{R}$). The retention of the application always corresponds to the Pi type of the function being applied.

Let This rule requires that the newly introduced definition d is well formed according to the corresponding rules in Section 5.5.2.2.

Conv Conversion (Figure 5.16) is defined as a reflexive symmetric transitive closure of reduction (Section 5.4).

5.5.2.2 Definitions

Definitions are checked using rules in Figure 5.17.

DefsBase and DefsStep check sequences of definitions. Sequences of definitions are telescopic, which means that the definition d_i is added to the environment before checking $d_{i+1}, \ldots, d_n$.

DefAbstr says that abstract definitions (those that do not have a body) need to have their type checked in an erased context, and that is all.

DefTerm also checks that the given term has the asserted type in the asserted context.

DefClauses checks all clauses in a definition. It (repeatedly) invokes $\textsc{Clause}^n_r$, where the superscript n is the name of the (whole) definition. This name must appear on the LHS of each clause.

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

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

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

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

Figure 5.15: Type and erasure checking rules for $\mathsf{TT}^{\mathsf{RE}}_{\star}$ terms

$$\frac{\Gamma \vdash \sigma \rightsquigarrow \tau}{\Gamma \vdash \sigma \approx \tau}\ \textsc{ConvRed}$$

$$\frac{}{\Gamma \vdash \tau \approx \tau}\ \textsc{ConvRefl} \qquad \frac{\Gamma \vdash \sigma \approx \tau}{\Gamma \vdash \tau \approx \sigma}\ \textsc{ConvSym} \qquad \frac{\Gamma \vdash \tau \approx \sigma \qquad \Gamma \vdash \sigma \approx \rho}{\Gamma \vdash \tau \approx \rho}\ \textsc{ConvTrans}$$

Figure 5.16: Conversion rules of $\mathsf{TT}^{\mathsf{RE}}_{\star}$

$$\frac{}{\Gamma \vdash \textsc{Defs}(\emptyset)}\ \textsc{DefsBase} \qquad \frac{\Gamma \vdash \textsc{Def}(d_1) \qquad \Gamma, d_1 \vdash \textsc{Defs}(d_2, \ldots, d_n)}{\Gamma \vdash \textsc{Defs}(d_1, d_2, \ldots, d_n)}\ \textsc{DefsStep}$$

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

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

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

FIGURE 5.17: Type and erasure checking rules for $\mathsf{TT}^{\mathsf{RE}}_\star$ definitions

Pattern clauses Rule CLAUSE checks individual clauses. The conclusion of CLAUSE is more restrictive than the general syntax of pattern clauses:

$$\Gamma \vdash \textsc{Clause}_r^f\left(\Pi.\ \lfloor f \rfloor \overline{\widehat{s}P} = R\right),$$

It requires a LHS in the form $\lfloor f \rfloor \overline{\widehat{s}P}$, where the name of the function, expressed as $\lfloor f \rfloor$, is applied to zero or more patterns.

Rule CLAUSE furthermore requires that Π is a well formed environment, that $\lfloor f \rfloor \overline{\widehat{s}P}$ is a well-formed LHS, that free pattern variables on the LHS are exactly the variables bound in Π, that the LHS is linear in pattern variables, and that it is forced-pattern-consistent.

Finally, the type check itself is performed using the approach established in Idris [Bra13], which has the advantage that it does not require unification or any other complicated (and potentially untrustworthy) procedures, while retaining the full power of dependent pattern matching. We check the LHS as a pattern and the RHS as a term, and then verify that their types are convertible. This approach was further discussed in Section 2.1.8.2.

5.5.2.3 Patterns

Patterns are checked using the rules in Figure 5.20. The general form of the pattern typing judgement is $\Gamma; \Pi \vdash P :_r^q \tau$, where

- Γ is the usual typing environment;
- Π is the pattern variable environment of the current clause;

$$
\begin{aligned}
\mathrm{FPV}_\Pi(n) &= \emptyset && \text{if } n \notin \mathrm{BV}(\Pi)\\
\mathrm{FPV}_\Pi(n) &= \{n\} && \text{if } n \in \mathrm{BV}(\Pi)\\
\mathrm{FPV}_\Pi(\lceil T \rceil) &= \emptyset\\
\mathrm{FPV}_\Pi(\lfloor f \rfloor) &= \emptyset\\
\mathrm{FPV}_\Pi(\lceil c \rceil) &= \emptyset\\
\mathrm{FPV}_\Pi(F \mathbin{\widehat{r}} X) &= \mathrm{FPV}_\Pi(F) \cup \mathrm{FPV}_\Pi(X)
\end{aligned}
$$

Figure 5.18: Free pattern variables

```
data Erased : Type → Type where
  Poof : (a : Type) → (x :E a) → Erased a

f : Erased a → a
(a : Type) (x :R a)
  f (Poof a x) = x
```

Figure 5.19: Example of an erasure-incorrect program

- P is the pattern;
- τ is the type of the pattern;
- r is the retention of the pattern (its context).
- q is the retention of the (whole) function containing the pattern. This is used to determine retention for constructor matches since a (non-forced) constructor *always* inspects (results in a runtime constructor tag check), unless in an entirely erased function.

Observation 5.1. If we start checking a clause with Clause, all pattern checking rules involved in the process are always invoked with $r \leqslant q$.

In Idris, patterns are checked simply as terms [Bra13], but in the presence of "subtyping" (Lemma 5.11), we need to be more careful about the direction of data flow. For example, consider the program in Figure 5.19, especially noting the annotation E on the second argument of Poof and the annotation R on the second pattern variable of f.

Using the *term* checking rules, we can verify that the LHS, f (Poof a x), successfully typechecks as a *term* with the type a. Since the RHS has the type a as well, this would be accepted as a valid pattern clause.

However, this does not make sense – we shouldn't allow having the second argument of Poof marked as erased, while being able to project it out of the constructor application.

The problem arises from the fact that the pattern variable $(x :_\mathsf{R} a)$ appears as the second argument to Poof, despite the erasability mismatch. This does make sense in terms – even if Poof does not use its second argument, we should surely be able to

pass an unerased value in it. However, in patterns, data flow is reversed and data flows *into* pattern variables, rather than the other way around.

- Term (and pattern) variables in *terms* are data sources; pattern variables in *patterns* are data sinks.
- Usage of data sources is bounded from below by how data sinks use the data flowing in.

Then we can say that while the term checking rule Ref required that any variable appearing in a runtime context must be retained, in patterns, we have to require the opposite – that if a pattern variable is needed at runtime (which means that it is used at runtime on the RHS), it makes the whole pattern needed at runtime.

In the typing rules, we go even farther and we require that pattern variables are bound *exactly* with the retention of the context in which they appear in the pattern[2].

PatVar requires that the pattern variable is bound with the *same* retention as the context in which it appears in the pattern. The reason is explained above.

PatCtor Rule PatCtor says that anytime a constructor is matched in a definition bound with retention q, then the constructor must be bound at least with retention q and the retention of the surrounding pattern must be at least q.

PatApp is similar to the rule App for terms.

PatDefName checks that for pattern $\lfloor f \rfloor$, the name f appears in the context and uses its type as the type of $\lfloor f \rfloor$. All erasure annotations are equal to q, the retention of the whole definition, since $\lfloor f \rfloor$ is at the root of the pattern.

PatForced checks forced patterns as terms in the environment Γ, Π, which no longer keeps track of the division between Γ and Π (unlike the pair $\Gamma; \Pi$; see Remark 5.3).

PatForcedCtor A forced constructor is checked like other forced patterns.

PatConv Finally, the pattern conversion rule PatConv is the pattern variant of the term conversion rule Conv.

5.5.2.4 Substitutions

Definition 5.12 (Well-typed substitution)**.** Substitution μ is well-typed on the set of names S, written $\Gamma \vdash \mu :^S \Pi$, if it substitutes pattern variables for terms of the corresponding types, i.e. if the following rule SubstWT holds.

$$\frac{n \in S \qquad (n :_s \sigma) \in \Pi}{\Gamma \vdash \mu(n) :_s \sigma[\mu]}\ \textsc{SubstWT}$$

For the special case where $\mathrm{Dom}(\mu) = \mathrm{BV}(\Pi)$ and $\Gamma \vdash \mu :^{\mathrm{BV}(\Pi)} \Pi$, we write $\Gamma \vdash \mu : \Pi$.

[2]This is needed for a prerequisite of Pattern Lemma, namely Lemma 5.48.

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

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

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

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

Figure 5.20: Pattern checking rules for $\mathsf{TT}^{\mathsf{RE}}_\star$

5.5.3 Unusual programs allowed by the rules

Matching on partially applied constructors The typing rules allow matching on partially applied constructors, as seen in Eisenberg's Pico or Haskell's type families [Eis16].

```
isLeft : (a → Either a a) → Bool
isLeft  Left   = True
isLeft  Right  = False
```

Since this might complicate compilation, implementations may want to disallow it.

Typecase Types are terms and thus the following program is allowed.

```
isBool : Type → Bool
isBool  Bool  = True
isBool  _     = False
```

It is also interesting that the first argument of isBool is *unerased*, despite having type Type.

The above program works because $\mathsf{TT}_\star$ does not make a distinction between type constructors and data constructors and does not keep track of their grouping into type families – they are simply bound as **constructor**s using **let**.

Pattern matching on $\mathsf{TT}_\star$ does not define matching on binders, especially Pi. Adding support for binders might be useful for typecase.

Non-uniform columns The following function is accepted by $\mathsf{TT}_\star$.

$$
\begin{array}{llll}
\mathsf{notId} : (a : \mathsf{Type}) \to (x : a) \to a \\
\mathsf{notId} & [\mathsf{Bool}] & \mathsf{True} & = \mathsf{False} \\
\mathsf{notId} & [\mathsf{Bool}] & \mathsf{False} & = \mathsf{True} \\
\mathsf{notId} & [\mathbb{N}] & \mathsf{Z} & = \mathsf{S}\ \mathsf{Z} \\
\mathsf{notId} & [\mathbb{N}] & (\mathsf{S}\ n) & = \mathsf{Z} \\
\mathsf{notId} & a & x & = x
\end{array}
$$

Even though the function "looks like" performing typecase, the type argument a is erased. The function works by matching constructors of various type families with a value of an unknown type – which might not even be an inductive type, such as in $\mathsf{notId}\ (\mathbb{N} \to \mathbb{N})\ (\lambda x : \mathbb{N}.\ x)$. Implementations may want to disallow this behaviour.

5.5.4 Forced patterns

Definition 5.13. We say that a LHS of a pattern clause $\lfloor f \rfloor_{\widehat{s}} \overline{X}$ is *forced-pattern-consistent* if the following holds.

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

I do not give an algorithm to check whether forced patterns are consistent; I expect the coverage checker to verify it (Section 5.3).

This property is related to *respectfulness of patterns*, as defined by Goguen, McBride, and McKinna [GMM06].

5.6 Erasure

The erasure translation is shown in Figure 5.21. Erasure removes all parts of programs annotated as erasable with E, replaces types (terms to the right of each colon) with $\Box$, and replaces erasure annotations with $\bullet$.

The presentation of the erasure translation relies on the notation introduced in Section 5.2.2, which allows writing $(\lambda x.\ M)$ instead of $(\lambda x :_\bullet \Box = \textbf{variable}.\ M)$, and similar shorthands.

5.7 Metatheory

This section establishes some metatheory of $\mathsf{TT}_\star$.

The development of metatheory has heavily influenced and simplified the design of the calculus and its pattern matching facilities. Most notably, I originally presented the calculus with case trees as the pattern matching primitive. However, I found the presentation easier with pattern clauses, and I deferred case trees to an extension

Abstract definition bodies:

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

Pattern clauses:

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

Substitutions

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

Terms:

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

Patterns:

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

Environments and telescopes of definitions:

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

Figure 5.21: Erasure translation, removing erasable code

(Section 7.2), although there are practical advantages to using case trees in the core language (Section 7.2.1).

The main soundness result is Theorem 5.2, which states that erasure commutes with (single-step) reduction. Assuming the Church-Rosser property (Conjecture 5.1), I extend this result to multi-step reduction:

- I prove the Pattern Lemma (Lemma 5.50), which says that values coming from pattern matching have correct types. This is a key step towards subject reduction of pattern matching.
- I prove Subject Reduction (Theorem 5.1).
- In Corollary 5.57, I prove that erasure commutes with multi-step reduction. In particular, if a program reduces to a ground term (a term that erases to itself), the erased program reduces to *the same* term.

5.7.1 Basic definitions

5.7.1.1 Environments

Definition 5.14 (Well-formed environments)**.** Environment Γ is *well-formed* iff the following holds according to the rules defined in Figure 5.17.

$$\vdash \textsc{Defs}(\Gamma)$$

A pair of environments $\Gamma; \Pi$ is well formed if $\vdash \textsc{Defs}(\Gamma)$ and $\Gamma \vdash \textsc{Defs}(\Pi)$.

Remark 5.4. An environment is a sequence of definitions, as defined by its syntax in Figure 5.2. By the definition of Defs, names defined earlier in the sequence are available in definitions later in the sequence (but not vice versa); such sequences are called *telescopic*.

Definition 5.15. We write $d \in \Gamma$ to express that environment Γ contains definition d.

$$\frac{}{d \in \Gamma, d}\ \textsc{ElemBase} \qquad \frac{d \in \Gamma}{d \in \Gamma, d'}\ \textsc{ElemStep}$$

5.7.1.2 Variables and substitution

Definition 5.16 (Free variables)**.** We write $\mathrm{FV}(T)$ for the set of free variables in term T. We use $\mathrm{FV}(-)$ also for non-terms, such as patterns or definitions.

Definition 5.17 (r-bound names)**.** We say that name n is r-bound in environment Γ if

$$(n :_r \tau = b) \in \Gamma$$

for some τ and b.

Definition 5.18 (R-bound variables)**.** We write $\mathrm{RBV}(\Gamma)$ for the set of names R-bound in environment Γ.

$$n \in \mathrm{RBV}(\Gamma) \quad \Leftrightarrow \quad (n :_{\mathsf{R}} \tau = b) \in \Gamma \text{ for any } \tau \text{ and } b$$

Definition 5.19 (Substitution in substitutions)**.** Substitution can be defined for substitution functions as follows.

$$\big(\mu[n \mapsto N]\big)(m) := \mu(m)[n \mapsto N]$$

5.7.2 General properties

Lemma 5.1 (Weakening and reduction)**.** *Adding a definition to an environment does not break existing reductions.*

$$\frac{\Gamma \vdash T \rightsquigarrow T'}{\Gamma, d \vdash T \rightsquigarrow T'}\ \textit{WeakRed}$$

Proof. By induction on the derivation of reduction, using disjointness of binders. □

Lemma 5.2 (Weakening and conversion)**.** *Adding a definition to an environment does not break existing conversions.*

$$\frac{\Gamma \vdash T \approx T'}{\Gamma, d \vdash T \approx T'}\ \textit{WeakConv}$$

Proof. By induction on the derivation of conversion, using Lemma 5.1 in ConvRed. □

Lemma 5.3 (Weakening)**.** *Adding a definition to an environment does not break existing type derivations.*

$$\frac{\Gamma \vdash T :_r \tau}{\Gamma, d \vdash T :_r \tau}\ \textit{Weakening}$$

Proof. By induction on the proof derivation, using the convention that binders bind distinct names (Section 5.3). For the rule Conv, we use Lemma 5.2. □

Lemma 5.4 (Thinning)**.** *Adding multiple definitions to an environment does not break existing derivations.*

$$\frac{\Gamma \vdash T \rightsquigarrow T'}{\Gamma, \Gamma' \vdash T \rightsquigarrow T'} \qquad \frac{\Gamma \vdash T \approx T'}{\Gamma, \Gamma' \vdash T \approx T'} \qquad \frac{\Gamma \vdash T :_r \tau}{\Gamma, \Gamma' \vdash T :_r \tau}$$

Proof. By induction on the size of Γ', using the appropriate weakening lemmas. □

Lemma 5.5 (Thinning II.)**.** *Adding multiple definitions in the middle of an environment does not break existing reduction/type/conversion derivations.*

$$\frac{\Gamma, \Pi \vdash T \rightsquigarrow T'}{\Gamma, \Gamma', \Pi \vdash T \rightsquigarrow T'} \qquad \frac{\Gamma, \Pi \vdash \tau \approx \tau'}{\Gamma, \Gamma', \Pi \vdash \tau \approx \tau'} \qquad \frac{\Gamma, \Pi \vdash T :_r \tau}{\Gamma, \Gamma', \Pi \vdash T :_r \tau}$$

Proof. By induction on the proof derivation, using the convention that binders bind distinct names (Section 5.3). □

Lemma 5.6.

$$\frac{\Gamma \vdash \textit{Defs}(\Gamma')}{\mathrm{FV}(\Gamma') \subseteq \mathrm{BV}(\Gamma)}$$

Proof. By induction on the size of Γ', mutually with Lemmas 5.8 and 5.9, using disjointness of binders and thinning.

In the inductive step, we observe that free variables can come from the type and from the body of each definition and that both type and body are well-typed in their respective environments. We use Lemmas 5.8 and 5.9 to observe that for every definition, pattern variables and self-references are the only new names introduced. These names are however bound immediately in the definition. □

Corollary 5.7.

$$\frac{\vdash \textit{Defs}(\Gamma) \qquad (n :_s \sigma = b) \in \Gamma}{\mathrm{FV}(\sigma) \subseteq \mathrm{BV}(\Gamma) \qquad \mathrm{FV}(b) \subseteq \mathrm{BV}(\Gamma)}$$

Proof. By well-formedness of Γ, this definition is well-typed in some prefix of Γ, which we call Δ. By Lemma 5.6, we have $\mathrm{FV}(n :_s \sigma = b) \subseteq \mathrm{BV}(\Delta) \subseteq \mathrm{BV}(\Gamma)$. □

Lemma 5.8 (Free variables of terms and their types)**.** *Let Γ be a well formed environment, and let T be a term such that $\Gamma \vdash T :_r \tau$. Then*

$$\mathrm{FV}(T) \subseteq \mathrm{BV}(\Gamma)$$
$$\mathrm{FV}(\tau) \subseteq \mathrm{BV}(\Gamma)$$

Proof. By induction on the typing derivation of T, mutually with Lemma 5.6. The only interesting cases are:

Ref with $T = n$ where n is a name and its type was derived using Ref. By Corollary 5.7, we obtain $\mathrm{FV}(\tau) \subseteq \mathrm{BV}(\Gamma)$.

Conv with $\Gamma \vdash T :_r \sigma$ where $\Gamma \vdash \tau :_{\mathsf{E}} \mathsf{Type}$ and $\Gamma \vdash \sigma \approx \tau$. We apply the induction hypothesis to well-typedness of τ.

□

Lemma 5.9 (Free variables of patterns and their types)**.** *Let $\Gamma; \Pi$ be a well formed pair of environments, and let P be a pattern such that $\Gamma; \Pi \vdash T :_r^q \tau$. Then*

$$\mathrm{FV}(P) \subseteq \mathrm{BV}(\Gamma, \Pi)$$
$$\mathrm{FV}(\tau) \subseteq \mathrm{BV}(\Gamma, \Pi)$$

Proof. Analogously to Lemma 5.8. □

Lemma 5.10 (Reduction preserves well-scoping)**.**

$$\frac{\vdash \mathit{Defs}(\Gamma) \qquad \Gamma \vdash T \rightsquigarrow T' \qquad \mathrm{FV}(T) \subseteq \mathrm{BV}(\Gamma)}{\mathrm{FV}(T') \subseteq \mathrm{BV}(\Gamma)}$$

Proof. By induction on the derivation of reduction. The only interesting cases are:

RedVar, where we use well-formedness of Γ and Lemma 5.8;

RedClauses, where $\mathrm{Dom}(\mu) = \mathrm{FPV}_{\Pi_k}(L_k)$ and $\mathrm{FV}(\mu) \subseteq \mathrm{FV}(f\,\overline{\widehat{r}\,X}) = \mathrm{FV}(T)$. Finally, $\mathrm{FV}(R_k) \subseteq \mathrm{BV}(\Gamma, \Pi_k)$ because f is bound in Γ, which is well-formed, and we can thus use Lemma 5.8.

□

Remark 5.5. It is not the case that

$$\frac{\vdash \mathrm{Defs}(\Gamma) \qquad \Gamma \vdash T \rightsquigarrow T' \qquad \mathrm{FV}(T') \subseteq \mathrm{BV}(\Gamma)}{\mathrm{FV}(T) \subseteq \mathrm{BV}(\Gamma)}$$

Consider $T = (\lambda x.\ \lambda y.\ y)\ z$, which reduces to a closed term regardless of whether z is bound in Γ or not.

This is why we postpone Lemma 5.29 after Church-Rosser (Conjecture 5.1).

Lemma 5.11.

$$\frac{\Gamma \vdash T :_s \tau \qquad r \leqslant s}{\Gamma \vdash T :_r \tau}$$

Proof. By induction on the proof derivation. The only interesting case is Ref, where the relation $r \leqslant s$ is satisfied. □

Lemma 5.12 (Coherence)**.** *If τ is a type of a term, it has itself type Type.*

$$\frac{\vdash \mathit{Defs}(\Gamma) \qquad \Gamma \vdash T :_r \tau}{\Gamma \vdash \tau :_E \mathsf{Type}}$$

Proof. By induction on the typing derivation and the size of Γ. □

5.7.3 Substitution

Lemma 5.13.

$$\frac{m \notin \mathrm{FV}(N)}{T[m \mapsto M][n \mapsto N] = T[n \mapsto N][m \mapsto M[n \mapsto N]]}$$

Proof. By induction on the structure of T. □

Lemma 5.14 (Commutativity of substitution)**.**

$$\frac{m \notin \mathrm{FV}(N) \qquad n \notin \mathrm{FV}(M)}{T[m \mapsto M][n \mapsto N] = T[n \mapsto N][m \mapsto M]}$$

Proof. From Lemma 5.13, observing that $M[n \mapsto N] = M$. □

5.7.4 Pattern matching and reduction

Lemma 5.15 (Substitution preserves well-formedness of patterns)**.**

$$\frac{\Gamma, n :_s \sigma, \Gamma'; \Pi \vdash \mathit{PatWF}(P)}{\Gamma, \Gamma'[n \mapsto N]; \Pi[n \mapsto N] \vdash \mathit{PatWF}\big(P[n \mapsto N]\big)}$$

Proof. By induction on the derivation of well-formedness because substitution for (non-pattern) variables does not interfere with the requirements of well-formedness of patterns – these depend only on non-variables. □

Lemma 5.16. *Substitution for variables preserves pattern match.*

$$\frac{\Gamma, n :_s \sigma, \Gamma'; \Pi \vdash P \parallel_\mu T}{\Gamma, \Gamma'[n \mapsto N]; \Pi[n \mapsto N] \vdash P[n \mapsto N] \parallel_{\mu[n \mapsto N]} T[n \mapsto N]}$$

Proof. By induction on the derivation of pattern match, using the fact that $n \mapsto N$ substitutes for a variable and therefore constructors and function names remain unaffected. □

Lemma 5.17. *Substitution for variables preserves pattern mismatch.*

$$\frac{\Gamma, n :_s \sigma, \Gamma' \vdash P \not\parallel T}{\Gamma, \Gamma'[n \mapsto N] \vdash P[n \mapsto N] \not\parallel T[n \mapsto N]}$$

Proof. By induction on the derivation of mismatch, using the fact that $n \mapsto N$ substitutes for a variable and therefore constructors and function names remain unaffected. □

5.7.5 Reduction

Lemma 5.18 (Thickening for reduction)**.**

$$\frac{\vdash \textsc{Defs}(\Gamma, \Delta, \Gamma') \qquad \Gamma, \Delta, \Gamma' \vdash T \rightsquigarrow T' \qquad \mathrm{FV}(\Gamma') \cap \mathrm{BV}(\Delta) = \emptyset \qquad \mathrm{FV}(T) \cap \mathrm{BV}(\Delta) = \emptyset}{\Gamma, \Gamma' \vdash T \rightsquigarrow T'}$$

The environment Γ' *can also be empty.*

Proof. By induction on the derivation of reduction. The interesting cases are RedVar and RedClauses, where disjointness of $\mathrm{FV}(T)$ and $\mathrm{BV}(\Delta)$ is preserved thanks to disjointness of $\mathrm{FV}(\Gamma')$ and $\mathrm{BV}(\Delta)$ and well-typedness of the environments. □

Lemma 5.19 (Thickening for repeated reduction)**.**

$$\frac{\vdash \textsc{Defs}(\Gamma, \Delta, \Gamma') \qquad \Gamma, \Delta, \Gamma' \vdash T \rightsquigarrow^\star T' \qquad \mathrm{FV}(\Gamma') \cap \mathrm{BV}(\Delta) = \emptyset \qquad \mathrm{FV}(T) \cap \mathrm{BV}(\Delta) = \emptyset}{\Gamma, \Gamma' \vdash T \rightsquigarrow^\star T'}$$

The environment Γ' *can also be empty.*

Proof. By induction on the length of the reduction sequence, using Lemma 5.18 in every step, along with Lemma 5.10 to carry over disjointness of $\mathrm{BV}(\Gamma)$ and $\mathrm{FV}(T)$ along reduction arrows. □

Lemma 5.20 (Reduction commutes with substitution)**.**

$$\frac{\vdash \textsc{Defs}(\Gamma, n :_s \sigma, \Gamma') \qquad \Gamma, n :_s \sigma, \Gamma' \vdash T \rightsquigarrow T'}{\Gamma, \Gamma'[n \mapsto N] \vdash T[n \mapsto N] \rightsquigarrow T'[n \mapsto N]}$$

Proof. By induction on the derivation of reduction. The structural rules are solved by straightforward induction; let us discuss the computation rules.

RedVar Here necessarily $T = m \neq n$ because n is bound as a variable, where RedVar would not apply. By inversion of RedVar, we have $(m :_t \tau = T') \in \Gamma, \Gamma'$ and by disjointness of binders, m is bound either in Γ or in Γ'.

If m is bound in Γ, then by well-formedness of Γ, Corollary 5.7 and disjointness of binders, $\mathrm{FV}(T') \subseteq \mathrm{BV}(\Gamma)$. Hence $T[n \mapsto N] = T = m$ and $T'[n \mapsto N] = T'$ and we can use Thickening (Lemma 5.18) to show $\Gamma \vdash T[n \mapsto N] \rightsquigarrow T'[n \mapsto N]$. We close by Thinning (Lemma 5.4) to obtain $\Gamma, \Gamma'[n \mapsto N] \vdash T[n \mapsto N] \rightsquigarrow T'[n \mapsto N]$.

If m is bound in Γ', then we obtain the result immediately by applying RedVar in the modified environment $\Gamma, \Gamma'[n \mapsto N]$.

Redex with $T = (\lambda x :_t \tau.\ M)\,\widehat{r}\ X$ and $T' = M[x \mapsto X]$. By disjointness of binders, $x \notin \mathrm{FV}(N)$ and by Lemma 5.13, $M[n \mapsto N][x \mapsto X[n \mapsto N]] = M[x \mapsto X][n \mapsto N]$.

RedClauses with $T = f\,\widehat{r}\,\overline{X}$. Assuming disjointness of binders, $n \neq f$ because n binds a **variable** and RedClauses would not be applicable otherwise. Only two hypotheses of RedClauses mention $\overline{X}$, and thus could possibly be affected by the substitution $[n \mapsto N]$.

We know that $\Gamma, n :_s \sigma, \Gamma'; \Pi_k \vdash L_k \parallel_\mu f\,\widehat{r}\,\overline{X}$ and we need to show that there is μ' such that $\Gamma[n \mapsto N], \Gamma'[n \mapsto N]; \Pi_k[n \mapsto N] \vdash L_k[n \mapsto N] \parallel_{\mu'} (f\,\widehat{r}\,\overline{X})[n \mapsto N]$, and thus we take $\mu' := \mu[n \mapsto N]$. By Lemma 5.16 and well-formedness of L_k provided by a hypothesis of RedClauses, the pattern still matches.

We also need to show that mismatch in clauses before C_k is preserved by substitution. This follows from Lemma 5.17 and disjointness of binders.

The reduction rule is therefore applicable after substitution and we can observe that we obtain the required result: $R_k[n \mapsto N][\mu'] = R_k[n \mapsto N][\mu[n \mapsto N]] = R_k[\mu][n \mapsto N]$ by Lemma 5.13 and disjointness of binders.

RedLetElim, RedLetAppL, RedLetAppR By induction, assuming disjointness of binders.

□

5.7.5.1 Mismatch

Lemma 5.21.

$$\frac{\Gamma; \Pi \vdash c \widehat{\tau} \overline{X} \parallel_\mu c' \widehat{\tau} \overline{X'} \qquad \textit{Dom}(\mu) \cap \text{BV}(\Gamma) = \emptyset \qquad c, c' \textit{ constructors in } \Gamma}{c = c'}$$

Proof. By induction on the length of the sequence $\overline{X}$, using MatchCtor in the base case and MatchApp in the induction step because these are the only rules that could possibly apply. □

Lemma 5.22.

$$\frac{\Gamma; \Pi \vdash c \widehat{\tau} \overline{X} \parallel_\mu c' \widehat{\tau} \overline{X'} \qquad \textit{Dom}(\mu) \cap \text{BV}(\Gamma) = \emptyset \qquad c, c' \textit{ constructors in } \Gamma}{X_i \parallel_\mu X_i'}$$

$$\frac{\Gamma; \Pi \vdash \lceil c \rceil \widehat{\tau} \overline{X} \parallel_\mu c' \widehat{\tau} \overline{X'} \qquad \textit{Dom}(\mu) \cap \text{BV}(\Gamma) = \emptyset \qquad c, c' \textit{ constructors in } \Gamma}{X_i \parallel_\mu X_i'}$$

Proof. Let us express $\overline{X} = (\overline{Y}, X_i, \overline{Z})$, and let's do the same for X'. Then we proceed by induction on the length of the sequence $\overline{Z}$, using MatchApp in both base case and the inductive step. □

Lemma 5.23 (Soundness of mismatch)**.**

$$\frac{\Gamma \vdash P \not\parallel T \qquad \textit{Dom}(\mu) \cap \text{BV}(\Gamma) = \emptyset}{\Gamma; \Pi \nvdash P \parallel_\mu T}$$

Proof. By induction on the derivation of mismatch.

MismatchHead $P = c \widehat{\tau} \overline{X}$ and $T = c' \widehat{\tau'} \overline{X'}$, where $c \neq c'$ are constructors bound in Γ. Assume $\Gamma; \Pi \vdash c \widehat{\tau} X \parallel_\mu c' \widehat{\tau'} X'$. By Lemma 5.21, $c = c'$, which is a contradiction.

MismatchArg $P = c \widehat{\tau} \overline{X}$ and $T = c \widehat{\tau'} \overline{X'}$, where c is a constructor bound in Γ. By inversion of MismatchArg, $\Gamma \vdash X_i \not\parallel X_i'$ for some i. By induction, $\Gamma; \Pi \nvdash X_i \parallel_\mu X_i'$. Assume $\Gamma; \Pi \vdash P \parallel_\mu T$. By Lemma 5.22, $\Gamma; \Pi \vdash X_i \parallel_\mu X_i'$, which is a contradiction.

MismatchArgForced Like MismatchArg.

MismatchLHS Like MismatchArg.

□

Lemma 5.24 (Reduction preserves applications of constructors)**.**

$$\frac{\Gamma \vdash c \widehat{\tau} \overline{X} \rightsquigarrow T' \qquad (c :_s \sigma = \textbf{\textit{constructor}}) \in \Gamma}{T' = c \widehat{\tau} \overline{X'} \textit{ for some } X'}$$

Proof. By induction on the length of $\overline{X}$, we can show that reduction must occur in (exactly) one of the arguments X_i, because there are no reduction rules that could replace constructor c at the head. □

Lemma 5.25 (Reduction preserves mismatch)**.**

$$\frac{\Gamma \vdash P \not\parallel T \qquad \Gamma \vdash T \rightsquigarrow T'}{\Gamma \vdash P \not\parallel T'}$$

Proof. By induction on the derivation of mismatch.

MismatchHead By inversion of MismatchHead, we have $P = c\,\overline{\widehat{r}\,X}$ and $T = c'\,\overline{\widehat{r'}\,X'}$, where c and c' are different constructors in Γ. By Lemma 5.24, $T' = c'\overline{\widehat{r'}\,X''}$ and we can use MismatchHead to derive the required mismatch.

MismatchArg By inversion of MismatchArg, we have $P = c\,\overline{\widehat{r}\,X}$ and $T = c\,\overline{\widehat{r'}\,X'}$ where $\Gamma \vdash X_i \not\parallel X'_i$ for some i. Apply induction and MismatchArg.

MismatchArgForced Like MismatchArg.

MismatchLHS Like MismatchArg.

□

Lemma 5.26. *If we extend a matching substitution with another, disjoint one, the match is preserved.*

$$\frac{\mu \cap \mu' = \emptyset \qquad \Gamma; \Pi \vdash P \parallel_{\mu} T}{\Gamma; \Pi \vdash P \parallel_{\mu \cup \mu'} T}$$

Proof. By induction on the derivation of matching. We depend on the definition of pattern matching, where μ substitutes only for pattern variables. □

Lemma 5.27.

$$\frac{\Gamma; \Pi \vdash \lceil c \rceil\,\overline{\widehat{r}\,X} \parallel_{\mu} T \qquad (c :_s \sigma = \textbf{\textit{constructor}}) \in \Gamma}{T = c'\,\overline{\widehat{r'}\,X'} \qquad \textit{for some } \overline{r'} \textit{ and } \overline{X'} \qquad (c' :_{s'} \sigma' = \textbf{\textit{constructor}}) \in \Gamma}$$

Proof. By induction on the length of the spine of the application, inverting MatchApp in each step and MatchForcedCtor in the base case. □

5.7.5.2 Church-Rosser property

Conjecture 5.1 (Church-Rosser property)**.** *Reduction is confluent, i.e. if term T reduces to two (possibly different) terms M and M', then M and M' have a common reduct.*

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

5.7.6 Conversion

Lemma 5.28.

$$\frac{\Gamma \vdash T \approx T'}{\exists M. \quad \Gamma \vdash T \rightsquigarrow^\star M \mathrel{{}^\star\!\!\leftsquigarrow} T'}$$

Proof. By induction on the derivation of conversion, using Conjecture 5.1 □

Lemma 5.29 (Thickening for conversion)**.**

$$\frac{\vdash \textsc{Defs}(\Gamma, \Delta, \Gamma') \qquad \Gamma, \Delta, \Gamma' \vdash \tau \approx \tau' \qquad \mathrm{FV}(\tau) \cap \mathrm{BV}(\Delta) = \emptyset \qquad \mathrm{FV}(\tau') \cap \mathrm{BV}(\Delta) = \emptyset}{\Gamma, \Gamma' \vdash \tau \approx \tau'}$$

The environment Γ' *can also be empty.*

Proof. The approach from Lemma 5.18 will not work in cases like $\Gamma, \Delta, \Gamma' \vdash T \leftsquigarrow N \approx N' \rightsquigarrow T'$, where the disjointness property cannot be carried inwards from either side. Instead, we invoke Church-Rosser (Lemma 5.28) and observe that $\Gamma, \Delta, \Gamma' \vdash T \rightsquigarrow^\star M \mathrel{{}^\star\!\!\leftsquigarrow} T'$, where M is a common reduct of T and T' such that $\mathrm{FV}(M) \cap \mathrm{BV}(\Delta) = \emptyset$ by Lemma 5.10. Now we can use Lemma 5.19 to obtain $\Gamma, \Gamma' \vdash T \rightsquigarrow^\star M \mathrel{{}^\star\!\!\leftsquigarrow} T'$ and thus $\Gamma, \Gamma' \vdash T \approx T'$. □

Lemma 5.30 (Thickening for typing)**.**

$$\frac{\vdash \textsc{Defs}(\Gamma, \Delta, \Gamma') \qquad \Gamma, \Delta, \Gamma' \vdash T :_r \tau \qquad \mathrm{FV}(T) \cap \mathrm{BV}(\Delta) = \emptyset \qquad \mathrm{FV}(\tau) \cap \mathrm{BV}(\Delta) = \emptyset}{\Gamma, \Gamma' \vdash T :_r \tau}$$

$$\frac{\vdash \textsc{Defs}(\Gamma, \Delta, \Gamma') \qquad \Gamma, \Delta, \Gamma'; \Pi \vdash P :_r^q \tau \qquad \mathrm{FV}(P) \cap \mathrm{BV}(\Delta) = \emptyset \qquad \mathrm{FV}(\tau) \cap \mathrm{BV}(\Delta) = \emptyset \qquad \mathrm{FV}(\Pi) \cap \mathrm{BV}(\Delta) = \emptyset}{\Gamma, \Gamma' \vdash P :_r^q \tau}$$

$$\frac{\vdash \textsc{Defs}(\Gamma, \Delta, \Gamma') \qquad \Gamma, \Delta, \Gamma' \vdash \textsc{Defs}(\Pi) \qquad \mathrm{FV}(\Pi) \cap \mathrm{BV}(\Delta) = \emptyset}{\Gamma, \Gamma' \vdash \textsc{Defs}(\Pi)}$$

The environment Γ' *can also be empty.*

Proof. By induction on the typing derivation, using well-typedness of the environments and for Conv, Lemma 5.29. □

Lemma 5.31 (Conversion commutes with substitution)**.**

$$\frac{\vdash \textsc{Defs}(\Gamma, n :_s \sigma, \Gamma') \qquad \Gamma, n :_s \sigma, \Gamma' \vdash \tau \approx \tau'}{\Gamma, \Gamma'[n \mapsto N] \vdash \tau[n \mapsto N] \approx \tau'[n \mapsto N]}$$

Proof. By induction on the derivation of $\Gamma, n :_s \sigma, \Gamma' \vdash \tau \approx \tau'$. Occurrences of ConvRefl for $n \approx n$ are replaced with ConvRefl for $N \approx N$ and the step for ConvRed follows from Lemma 5.20. The rest is induction.

An alternative proof uses the Church-Rosser property (Lemma 5.28) together with Lemma 5.20. □

Lemma 5.32 (Substitution lemma)**.**

$$\frac{\vdash \mathit{Defs}(\Gamma, n :_s \sigma, \Gamma') \qquad \Gamma \vdash N :_{r \wedge s} \sigma \qquad \Gamma, n :_s \sigma, \Gamma' \vdash T :_r \tau}{\Gamma, \Gamma'[n \mapsto N] \vdash T[n \mapsto N] :_r \tau[n \mapsto N]}\ \mathit{Subst}$$

$$\frac{\vdash \mathit{Defs}(\Gamma, n :_s \sigma, \Gamma'; \Pi) \qquad \Gamma \vdash N :_{r \wedge s} \sigma \qquad \Gamma, n :_s \sigma, \Gamma'; \Pi \vdash P :^q_r \tau}{\Gamma, \Gamma'[n \mapsto N]; \Pi[n \mapsto N] \vdash P[n \mapsto N] :^q_r \tau[n \mapsto N]}\ \mathit{SubstPat}$$

$$\frac{\vdash \mathit{Defs}(\Gamma, n :_s \sigma, \Gamma') \qquad \Gamma \vdash N :_{r \wedge s} \sigma \qquad \Gamma, n :_s \sigma, \Gamma' \vdash \mathit{Def}(d)}{\Gamma, \Gamma'[n \mapsto N] \vdash \mathit{Def}\big(d[n \mapsto N]\big)}\ \mathit{SubstDef}$$

Proof. By (mutual) induction on the typing derivation of term T, pattern P, or definition d. Note that we do not need the Inversion Lemma here because we perform induction on the typing derivation, not the structure of T.

Terms **Axiom** Trivially.

Ref, where $T = m$. By disjointness of binders, $n \notin \mathrm{BV}(\Gamma)$ and from well-typedness of N, we obtain $\mathrm{FV}(\sigma) \subseteq \mathrm{BV}(\Gamma)$ and specifically $n \notin \mathrm{FV}(\sigma)$ by Lemma 5.8.

If $m \neq n$**,** then $T[n \mapsto N] = T = m$ and m is therefore bound either in Γ or Γ'.

If $(m :_t \tau = b) \in \Gamma$**,** then $r \leqslant t$ and by Lemma 5.8, $n \notin \mathrm{FV}(\tau)$. Thus $(m :_t \tau[n \mapsto N] = b) \in \Gamma$ and also $(m :_t \tau[n \mapsto N] = b) \in \Gamma, (n :_s \sigma), \Gamma$, and thus by Ref, we obtain the required result.

If $(m :_t \tau = b) \in \Gamma'$**,** then $(m :_t \tau[n \mapsto N] = b[n \mapsto N]) \in \Gamma'[n \mapsto N]$, and we can use Ref to obtain the required result.

If $m = n$**,** then by inversion of Ref, we have $\sigma = \tau$ and $r \leqslant s$. Since $r \leqslant s$, we have $r \wedge s = r$. From $\sigma = \tau$ and the above observation that $n \notin \mathrm{FV}(\sigma)$, we obtain $\tau[n \mapsto N] = \tau$. We can thus rewrite the assumption $\Gamma \vdash N :_{r \wedge s} \sigma$ as $\Gamma \vdash T[n \mapsto N] :_r \tau[n \mapsto N]$ and use Lemma 5.4 to obtain the required conclusion.

Lam Here, $T = \lambda m :_t \rho.\ M$. By inversion of Lam and induction, we have the following.

$$\frac{\Gamma, \Gamma'[n \mapsto N] \ \vdash\ \rho[n \mapsto N] :_{\mathsf{E}} \mathsf{Type}}{\Gamma, \Gamma'[n \mapsto N], m :_t \rho[n \mapsto N] \ \vdash\ M[n \mapsto N] :_r \tau[n \mapsto N]}$$

We can now use Lam to derive the desired conclusion.

Induction was possible because the context extension was well-typed, as per the requirement of Lam.

Pi Analogously to Lam.

App Here, $T = F \mathrel{\widehat{t}} X$. By induction, we have the following.

$$\Gamma, \Gamma'[n \mapsto N] \vdash F[n \mapsto N] :_r (x :_t \alpha[n \mapsto N]) \to \beta[n \mapsto N]$$
$$\Gamma, \Gamma'[n \mapsto N] \vdash X[n \mapsto N] :_{r \wedge t} \alpha[n \mapsto N]$$

By App, we obtain the following.

$$\Gamma, \Gamma'[n \mapsto N] \vdash (F \mathrel{\widehat{t}} X)[n \mapsto N] :_r \beta[n \mapsto N][x \mapsto X[n \mapsto N]]$$

By disjointness of binders, $x \notin \mathrm{FV}(N)$, and thus we can use Lemma 5.13 to obtain $\beta[n \mapsto N][x \mapsto X[n \mapsto N]] = \beta[x \mapsto X][n \mapsto N]$.

Let Here, $T =$ **let** d **in** M. Like in the cases for Lam and Pi, d is well-typed, which allows us to use induction in the context extended with d.

By induction, we obtain $\Gamma, \Gamma'[n \mapsto N], d[n \mapsto N] \vdash M[n \mapsto N] :_r \tau[n \mapsto N]$, and by (mutual) induction with SubstDef, we have $\Gamma, \Gamma'[n \mapsto N] \vdash \mathrm{Def}(d[n \mapsto N])$, which is exactly what we need to apply Let again.

Conv Use induction and Lemma 5.31.

Patterns By induction, analogously to terms.

Definitions Similarly by induction. The only interesting case is Clause, where we can observe that the set of free pattern variables does not change by substitution for n.

□

Lemma 5.33. *Well-typed multiple substitutions are idempotent.*

$$\frac{\vdash \mathit{Defs}(\Gamma) \qquad \Gamma \vdash \mathit{Defs}(\Pi) \qquad \Gamma \vdash \mu : \Pi}{\mu[\mu] = \mu}$$

Proof. By Definition 5.12 and Lemma 5.8, $\mathrm{Dom}(\mu) \cap \mathrm{FV}(\mu) = \emptyset$. □

Lemma 5.34 (Multiple substitution lemma)**.** *Let* Γ, Π *be a well formed environment. Then the result of a well-typed substitution is well-typed.*

$$\frac{\Gamma, \Pi \vdash T :_r \tau \qquad \Gamma \vdash \mu :^{\mathrm{FV}(T)} \Pi}{\Gamma \vdash T[\mu] :_r \tau[\mu]}\ \mathit{SubstMany}$$

Proof. By induction on the size of $\mathrm{BV}(\Pi) \cap \mathrm{FV}(T)$, using Lemma 5.32 together with Lemma 5.11 in every step. □

$$\frac{\Gamma \vdash \mathsf{Type} :_r \tau}{\Gamma \vdash \tau \approx \mathsf{Type}}\ \textsc{InvAxiom}$$

$$\frac{\Gamma \vdash n :_r \tau}{\exists b, s, \tau'. \qquad \Gamma \vdash \tau \approx \tau' \qquad (n :_s \tau' = b) \in \Gamma \qquad r \leqslant s}\ \textsc{InvRef}$$

$$\frac{\Gamma \vdash (\lambda n :_s \sigma.\, T) :_r \tau}{\exists \rho. \qquad \Gamma \vdash \tau \approx (n :_s \sigma) \to \rho \qquad \Gamma, n :_s \sigma \vdash T :_r \rho \qquad \Gamma \vdash \sigma :_{\mathsf{E}} \mathsf{Type}}\ \textsc{InvLam}$$

$$\frac{\Gamma \vdash \big((n :_s \sigma) \to \rho\big) :_r \tau}{\Gamma \vdash \tau \approx \mathsf{Type} \qquad \Gamma, n :_s \sigma \vdash \rho :_{\mathsf{E}} \mathsf{Type} \qquad \Gamma \vdash \sigma :_{\mathsf{E}} \mathsf{Type}}\ \textsc{InvPi}$$

$$\frac{\Gamma \vdash F \mathbin{\widehat{s}} X :_r \tau}{\exists \sigma, \rho, n. \qquad \Gamma \vdash \tau \approx \rho[n \mapsto X] \qquad \Gamma \vdash F :_r (n :_s \sigma) \to \rho \qquad \Gamma \vdash X :_{r \wedge s} \sigma}\ \textsc{InvApp}$$

$$\frac{\Gamma \vdash (\textbf{let}\ d\ \textbf{in}\ T) :_r \tau}{\Gamma \vdash \textsc{Def}(d) \qquad \Gamma, d \vdash T : \tau}\ \textsc{InvLet}$$

Figure 5.22: Components of Inversion lemma: terms

Remark 5.6. Lemma 5.34 is a bit weaker than it could be, since for every substituted name $(n :_s \sigma) \in \Pi$, it requires that $\Gamma \vdash \mu(n) :_s \sigma$, even though $\Gamma \vdash \mu(n) :_{r \wedge s} \sigma$ would be sufficient, by Lemma 5.32 – which is also why we need Lemma 5.11 in the proof. However, this formulation is more convenient for us.

Lemma 5.35 (Inversion lemma)**.** *Figures 5.22 and 5.23 show several admissible rules that form the inversion lemma, which says that we can invert the typing rules during structural induction, up to conversion.*

Proof. Instead of spelling out a proof for each rule separately, I give a general proof scheme.

For each rule, we proceed by induction on the typing derivation of the premise, which is generally $\Gamma \vdash M :_r \tau$; the term M varies between rules. There are two possibilities for the last rule in the typing derivation.

Conv By inversion of Conv, there is τ'' such that $\Gamma \vdash M :_r \tau''$ and $\Gamma \vdash \tau'' \approx \tau$. By induction, we obtain the conclusion of the inversion rule, which we name $C(\tau'')$.

We however need the conclusion $C(\tau)$. We therefore observe that in the conclusion of every inversion rule, $C(\tau)$, the type τ appears exactly once: in a conversion judgement $\Gamma \vdash \tau \approx \tau'$ for some right-hand side τ'.

From $C(\tau'')$, we have $\Gamma \vdash \tau'' \approx \tau'$, which we combine using ConvTrans and ConvSym with $\Gamma \vdash \tau'' \approx \tau$, to obtain $\Gamma \vdash \tau \approx \tau'$.

The desired right hand side τ' is the same in both $C(\tau)$ and $C(\tau'')$, and therefore the judgement $\Gamma \vdash \tau \approx \tau'$, together with the remaining judgements from $C(\tau'')$, gives us exactly the desired conclusion $C(\tau)$.

$$\frac{\Gamma;\Pi \vdash n :^{q}_{r} \tau}{\begin{array}{ll}\text{either} & \exists s. \quad (n :_s \sigma = \textbf{variable}) \in \Pi \quad s = r \quad \Gamma,\Pi \vdash \tau \approx \sigma \\ \text{or} & \exists s. \quad (n :_s \sigma = \textbf{constructor}) \in \Gamma \quad q \leqslant s \wedge r \quad \Gamma,\Pi \vdash \tau \approx \sigma\end{array}}\ \textsc{InvPatRef}$$

$$\frac{\Gamma;\Pi \vdash F \mathbin{\widehat{s}} X :^{q}_{r} \tau}{\begin{array}{ll}\exists \rho, \sigma, n. & \Gamma;\Pi \vdash F :^{q}_{r} (n :_s \sigma) \to \rho \quad \Gamma;\Pi \vdash X :^{q}_{r \wedge s} \sigma \\ & \Gamma,\Pi \vdash \tau \approx \rho[n \mapsto |X|]\end{array}}\ \textsc{InvPatApp}$$

$$\frac{\Gamma;\Pi \vdash [T] :^{q}_{r} \tau}{\Gamma,\Pi \vdash T :_r \tau}\ \textsc{InvPatForced}$$

$$\frac{\Gamma;\Pi \vdash \lfloor f \rfloor :^{q}_{r} \tau}{(f :_q \sigma = \overline{C}) \in \Gamma \quad q = r}\ \textsc{InvPatDefName}$$

$$\frac{\Gamma;\Pi \vdash \lceil c \rceil :^{q}_{r} \tau}{(c :_s \sigma = \textbf{constructor}) \in \Gamma \quad s \leqslant r \quad \Gamma,\Pi \vdash \sigma \approx \tau}\ \textsc{InvPatForcedCtor}$$

Figure 5.23: Components of Inversion lemma: patterns

Not Conv Since the conclusions of the typing rules (Figure 5.15) without Conv are syntactically disjoint, any typing judgement for a given term must be derived using the rule that corresponds to the (syntactic) form of the term.

For each term form, we can verify that the conclusions of the corresponding inversion lemma match the requirements of the corresponding typing rule.

We also use the reflexivity of conversion to obtain the extra judgement $\Gamma \vdash \tau \approx \tau'$ in the conclusion of each inversion lemma.

□

Remark 5.7. Since we assume disjointness of binders and the pair $\Gamma;\Pi$ appears in the assumption of InvPatRef, the two alternative conclusions of InvPatRef are indeed exclusive – n cannot be bound in both Γ and Π at the same time.

Lemma 5.36.

$$\frac{\Gamma \vdash \sigma \approx \sigma' \quad \Gamma, n :_s \sigma \vdash T :_r \tau}{\Gamma, n :_s \sigma' \vdash T :_r \tau}\ \textit{ConvCtx}$$

$$\frac{\Gamma \vdash \sigma \approx \sigma' \quad \Gamma, n :_s \sigma \vdash P :^{q}_{r} \tau}{\Gamma, n :_s \sigma' \vdash P :^{q}_{r} \tau}\ \textit{ConvCtxPat}$$

$$\frac{\Gamma \vdash \sigma \approx \sigma' \quad \Gamma, n :_s \sigma \vdash \mathit{Defs}(\Pi)}{\Gamma, n :_s \sigma' \vdash \mathit{Defs}(\Pi)}\ \textit{ConvCtxDef}$$

$$\frac{\Gamma \vdash \sigma \approx \sigma' \qquad \Gamma, n :_s \sigma \vdash T \approx T'}{\Gamma \vdash (\lambda n :_s \sigma.\ T) \approx (\lambda n :_s \sigma'.\ T')}\ \textsc{CongLam} \qquad \frac{\Gamma \vdash d \approx d' \qquad \Gamma, d \vdash T \approx T'}{\Gamma \vdash (\textbf{let}\ d\ \textbf{in}\ T) \approx (\textbf{let}\ d'\ \textbf{in}\ T')}\ \textsc{CongLet}$$

$$\frac{\Gamma \vdash \sigma \approx \sigma' \qquad \Gamma, n :_s \sigma \vdash \rho \approx \rho'}{\Gamma \vdash \big((n :_s \sigma) \to \rho\big) \approx \big((n :_s \sigma') \to \rho'\big)}\ \textsc{CongPi} \qquad \frac{\Gamma \vdash F \approx F' \qquad \Gamma \vdash X \approx X'}{\Gamma \vdash (F \mathrel{\widehat{s}} X) \approx (F' \mathrel{\widehat{s}} X')}\ \textsc{CongApp}$$

Figure 5.24: Congruence lemmas: terms

$$\frac{\Gamma \vdash \sigma \approx \sigma' \qquad \Gamma, (n :_s \sigma = b) \vdash b \approx b'}{\Gamma \vdash (n :_s \sigma = b) \approx (n :_s \sigma' = b')}\ \textsc{CongDef}$$

$$\frac{\Gamma \vdash \Pi \approx \Pi' \qquad \Gamma; \Pi \vdash L \approx L' \qquad \Gamma, \Pi \vdash R \approx R'}{\Gamma \vdash (\Pi.\ L = R) \approx (\Pi'.\ L' = R')}\ \textsc{CongClause}$$

$$\frac{\Gamma; \Pi \vdash F \approx F' \qquad \Gamma; \Pi \vdash X \approx X'}{\Gamma; \Pi \vdash (F \mathrel{\widehat{s}} X) \approx (F' \mathrel{\widehat{s}} X')}\ \textsc{CongPatApp} \qquad \frac{\Gamma, \Pi \vdash T \approx T'}{\Gamma; \Pi \vdash [T] \approx [T']}\ \textsc{CongPatForced}$$

Figure 5.25: Congruence lemmas: non-terms

Proof. By mutual induction on the derivations. The only interesting cases are Ref and PatRef, where we insert extra appeals to Conv and PatConv. □

Definition 5.20 (Conversion for non-terms)**.** In order to *state* that conversion is a congruence, we need to extend it to non-terms. Fortunately, since reduction ($\rightsquigarrow$) is already defined for non-terms, this is possible without any changes to the definition of conversion in Figure 5.16.

Lemma 5.37 (Conversion is a congruence)**.** *Figures 5.24 and 5.25 state sub-lemmas that establish conversion as a congruence.*

Proof. By mutual structural induction among the rules. I spell out the proof only for CongLam; all other sub-lemmas are proven analogously.

CongLam For a rule with two hypotheses, we prove the required property in two steps. First, we establish CongLamL, where only σ varies.

$$\frac{\Gamma \vdash \sigma \approx \sigma'}{\Gamma \vdash (\lambda n :_s \sigma.\ T) \approx (\lambda n :_s \sigma'.\ T)}\ \textsc{CongLamL}$$

We prove CongLamL by induction on the derivation of its hypothesis $\Gamma \vdash \sigma \approx \sigma'$. The last rule in the derivation can be one of the following.

$$\dfrac{\dfrac{\Gamma \vdash \sigma \approx \sigma'}{\Gamma \vdash (\lambda n :_s \sigma.\ T) \approx (\lambda n :_s \sigma'.\ T)}\ \textsc{CongLamL} \qquad \dfrac{\dfrac{\Gamma, n :_s \sigma \vdash T \approx T' \qquad \Gamma \vdash \sigma \approx \sigma'}{\Gamma, n :_s \sigma' \vdash T \approx T'}\ \textsc{ConvCtx}}{\Gamma \vdash (\lambda n :_s \sigma'.\ T) \approx (\lambda n :_s \sigma'.\ T')}\ \textsc{CongLamR'}}{\Gamma \vdash (\lambda n :_s \sigma.\ T) \approx (\lambda n :_s \sigma'.\ T')}\ \textsc{ConvTrans}$$

Figure 5.26: Proof of CongLam

ConvRefl We have $\sigma = \sigma'$ and we can re-apply ConvRefl to the whole term.

ConvRed By inversion, $\Gamma \vdash \sigma \rightsquigarrow \sigma'$. We apply RedLamL and re-apply ConvRed back to obtain the desired result.

ConvSym Apply induction and re-apply ConvSym.

ConvTrans Apply induction to both sub-derivations, re-apply ConvTrans.

Then we prove CongLamR', where only the term varies.

$$\dfrac{\Gamma, n :_s \sigma' \vdash T \approx T}{\Gamma \vdash (\lambda n :_s \sigma'.\ T) \approx (\lambda n :_s \sigma'.\ T')}\ \textsc{CongLamR'}$$

The proof is analogous to that for CongLamL; in the case for ConvRed, we use RedLamR.

Finally, we combine the components using ConvTrans and ConvCtx (Lemma 5.36) to derive the desired result, as shown in Figure 5.26.

In the proofs of all these sub-lemmas, we perform the same procedure as in the proof of CongLam – we derive the convertibility of larger terms from (inductively obtained) convertibility of their (direct) subterms by inspecting the subproofs and using the structurality of reduction. □

Remark 5.8. Mishra-Linger [ML08] postulates these rules in the definition of conversion. In $\mathsf{TT}_\star$, they are all admissible.

Lemma 5.38 (Conversion is substitutive)**.**

$$\dfrac{\Gamma \vdash N \approx N'}{\Gamma \vdash T[n \mapsto N] \approx T[n \mapsto N']}\ \textit{ConvSubst}$$

Proof. By induction on the structure of T. If $T = n$, use the hypothesis. Otherwise, use the congruence rules (Lemma 5.37) or ConvRefl. □

Lemma 5.39 (Renaming and substitution)**.**

$$\dfrac{n \notin \mathrm{FV}(T)}{T[n' \mapsto n][n \mapsto N] = T[n' \mapsto N]}$$

Proof. By induction on the structure of T. □

Lemma 5.40 (Pi and conversion)**.**

$$\frac{\Gamma \vdash \big((n :_s \sigma) \to \rho\big) \approx \big((n' :_{s'} \sigma') \to \rho'\big)}{n = n' \qquad s = s' \qquad \Gamma \vdash \sigma \approx \sigma' \qquad \Gamma, n :_s \sigma \vdash \rho \approx \rho'}$$

Proof. By the Church-Rosser property (Lemma 5.28), both sides must have a common reduct. The reduction rules admit only a reduct of the form $(n'' :_{s''} \sigma'') \to \rho''$, where $n = n'' = n'$, $s = s'' = s'$, $\Gamma \vdash \sigma \rightsquigarrow^\star \sigma'' \ {}^\star\!\!\leftsquigarrow \sigma'$, and therefore $\Gamma \vdash \sigma \approx \sigma'$. Furthermore, we must have $\Gamma, n :_s \sigma \vdash \rho \rightsquigarrow^\star \rho''$ and $\Gamma, n :_s \sigma' \vdash \rho'' \ {}^\star\!\!\leftsquigarrow \rho'$. By Lemma 5.36, we also have $\Gamma, n :_s \sigma \vdash \rho'' \ {}^\star\!\!\leftsquigarrow \rho'$, and thus $\Gamma, n :_s \sigma \vdash \rho \approx \rho'$. □

Lemma 5.41 (All types of the same term are convertible)**.** *Let Γ be a well formed environment and T be a term.*

$$\frac{\Gamma \vdash T :_r \tau \qquad \Gamma \vdash T :_r \tau'}{\Gamma \vdash \tau \approx \tau'} \ \textsc{UniqType}$$

Proof. By induction on the structure of T, using the inversion lemma (Lemma 5.35). We will also use transitivity and symmetry of conversion to allow ourselves to write chains of conversion judgements as $\Gamma \vdash \tau_1 \approx \ldots \approx \tau_k$, to reorder these chains arbitrarily, and to implicitly conclude $\Gamma \vdash \tau_1 \approx \tau_k$ from them.

If $T =$ Type, the result follows from InvAxiom, ConvTrans, and ConvSym.

If $T = n$, we apply InvRef to both typing derivations, from which we obtain $\Gamma \vdash \tau \approx \sigma \approx \tau'$, where σ is the type of the binding of n.

If $T = \lambda n :_s \sigma.\ M$, then by InvLam, we know that $\Gamma \vdash \tau \approx (n :_s \sigma) \to \rho$ and $\Gamma \vdash \tau' \approx (n :_s \sigma) \to \rho'$ for some types ρ, ρ'. By induction on M, we have $\Gamma, n :_s \sigma \vdash \rho \approx \rho'$, and by CongPi with ConvRefl, we obtain the missing link $\Gamma \vdash \big((n :_s \sigma) \to \rho\big) \approx \big((n :_s \sigma) \to \rho'\big)$.

If $T = (n :_s \sigma) \to \rho$, then by InvPi, $\Gamma \vdash \tau \approx \mathsf{Type} \approx \tau'$.

If $T = F \ \widehat{s}\ X$, we apply InvApp to both typing derivations to learn that $\Gamma \vdash \tau \approx \rho[n \mapsto X]$ for some ρ and n, and that $\Gamma \vdash \tau' \approx \rho'[n' \mapsto X]$ for some ρ' and n'. From the inversion lemma, we furthermore have:

$$\begin{aligned}
&\Gamma \vdash F :_r (n :_s \sigma) \to \rho \\
&\Gamma \vdash F :_r (n' :_s \sigma') \to \rho' \\
&\Gamma \vdash \big((n :_s \sigma) \to \rho\big) \approx \big((n' :_s \sigma') \to \rho'\big) \qquad \text{(induction on } F\text{)}
\end{aligned}$$

By Lemma 5.40 and Lemma 5.31, we obtain $\Gamma \vdash \rho[n \mapsto X] \approx \rho'[n \mapsto X]$, and thus by transitivity of conversion, $\Gamma \vdash \tau \approx \tau'$.

If $T =$ let d in T, the result follows directly from the inductive hypothesis.

□

5.7.7 Erasure

Lemma 5.42 (Erasure commutes with substitution)**.**

$$\langle T[x \mapsto X]\rangle = \langle T\rangle[x \mapsto \langle X\rangle]$$

Proof. By induction on the structure of T. □

Lemma 5.43 (Erasure commutes with substitution II.)**.** *Let T be a term and let μ be a substitution.*

$$\langle T[\mu]\rangle = \langle T\rangle[\langle \mu\rangle]$$

Proof. By induction on the structure of T. □

Lemma 5.44.

$$\frac{\Gamma;\Pi \vdash \textit{PatWF}_F(P)}{\Gamma;\Pi \vdash \textit{PatWF}_X(P)}$$

Proof. For all PatWF_F rules, there is an equivalent PatWF_X rule. □

Lemma 5.45. *Erasure preserves well-formedness of well-typed patterns in runtime definitions.*

$$\frac{\Gamma;\Pi \vdash \textit{PatWF}_\xi(P) \qquad \Gamma;\Pi \vdash P :^R_R \lambda}{\langle\Gamma\rangle;\Pi \vdash \textit{PatWF}_\xi\big(\langle P\rangle\big)}$$

where $\xi \in \{f, F, X\}$.

Proof. By induction on the proof derivation.

PatWF_f-Head with $P = \lfloor f \rfloor$.
By InvPatDefName, f is R-bound, its binding survives erasure and we can re-apply the rule after erasure.

PatWF_F-Ctor and PatWF_X-Ctor with $P = c$ bound as constructor in Γ.
By InvPatRef, c is R-bound, its binding survives erasure and we can re-apply the rule after erasure.

PatWF_F-ForcedCtor and PatWF_X-ForcedCtor with $P = \lceil c \rceil$ where c is a constructor in Γ. By InvPatForcedCtor, c is R-bound, its binding survives erasure and we can apply the rule after erasure.

PatWF_X-Forced with $P = [T]$.
Since $\langle P\rangle = [\langle T\rangle]$, we can re-apply the rule after erasure.

PatWF$_X$-Patvar with $P = n$ bound as variable in Π.

By InvPatRef, n is R-bound, its binding survives erasure and we can re-apply the rule after erasure.

PatWF$_\xi$-App with $P = F\ \widehat{s}\ X$.

By InvPatApp, $\Gamma; \Pi \vdash F :^{\mathsf{R}}_{\mathsf{R}} (x :_s \sigma) \to rho$ for some σ and ρ and $\Gamma; \Pi \vdash X :^{\mathsf{R}}_s \sigma$.
By induction, $\langle F \rangle$ is well-formed in Γ in the same way as F.

If $s = \mathsf{R}$, then we can apply induction to X to prove well-formedness of $\langle X \rangle$ and reapply PatWF$_\xi$-App after erasure.

If $s = \mathsf{E}$, then $\langle P \rangle = \langle F \rangle$.

If $\xi = f$, then induction yields $\Gamma; \Pi \vdash \textsc{PatWF}_f(\langle F \rangle)$, which is exactly what we need.

If $\xi = F$, then induction yields $\Gamma; \Pi \vdash \textsc{PatWF}_F(\langle F \rangle)$, which is exactly what we need, again.

If $\xi = X$, then induction yields $\Gamma; \Pi \vdash \textsc{PatWF}_F(\langle F \rangle)$, but we apply Lemma 5.44.

□

Lemma 5.46 (Variable survival)**.** *Free variables of an erased well-typed term are bound with R.*

$$\frac{\Gamma \vdash T :_R \tau \qquad n \in \mathrm{FV}(\langle T \rangle)}{n \in \mathrm{RBV}(\Gamma)}$$

Proof. By induction on the typing derivation. □

5.7.8 Pattern matching and types

Lemma 5.47.

$$\frac{\Gamma \vdash c\ \overline{\widehat{r}\ X} \approx c\ \overline{\widehat{r}\ X'} \qquad (c :_s \sigma = \textbf{\textit{constructor}}) \in \Gamma}{\forall i. \qquad \Gamma \vdash X_i \approx X'_i}$$

Proof. Since the application is headed by a constructor, and by Lemma 5.28, both sides reduce to $c\ \overline{\widehat{r}\ Y}$ for some $\overline{Y}$ such that $\Gamma \vdash X_i \rightsquigarrow^\star Y_i \ {}^\star\!\!\leftsquigarrow X'_i$ for each i. □

Lemma 5.48.

$$\frac{\Gamma; \Pi \vdash P :^q_r \tau}{\Gamma, \Pi \vdash |P| :_r \tau}$$

Proof. By induction on the typing derivation. The only interesting case is PatVar, where the strong assumptions of PatVar allow using Ref. Besides, Ref is also usable in the cases for PatCtor and PatDefName. □

Definition 5.21 (PWF)**.** To simplify the statement of the Pattern Lemma, we will pack the notions of well-formedness and forced-pattern-consistency together into a single property called PWF. These requirements are slightly different for constructor-headed and $\lfloor f \rfloor$-headed pattern applications.

$$\frac{\Gamma;\Pi \vdash \textsc{PatWF}_f\left(\lfloor f \rfloor \overline{\widehat{r} X}\right) \qquad \lfloor f \rfloor \overline{\widehat{r} X} \text{ is forced-pattern-consistent}}{\Gamma;\Pi \vdash \textsc{PWF}^T_\mu\left(\lfloor f \rfloor \overline{\widehat{r} X}\right)}\ \textsc{PWF-F}$$

$$\frac{\Gamma;\Pi \vdash \textsc{PatWF}_F\left(c \overline{\widehat{r} X}\right) \qquad \Gamma \vdash |c \overline{\widehat{r} X}|[\mu] \approx T \qquad (c :_s \sigma = \textbf{constructor}) \in \Gamma}{\Gamma;\Pi \vdash \textsc{PWF}^T_\mu(c \overline{\widehat{r} X})}\ \textsc{PWF-C}$$

$$\frac{\Gamma;\Pi \vdash \textsc{PatWF}_F\left(\lceil c \rceil \overline{\widehat{r} X}\right) \qquad \Gamma \vdash |\lceil c \rceil \overline{\widehat{r} X}|[\mu] \approx T \qquad (c :_s \sigma = \textbf{constructor}) \in \Gamma}{\Gamma;\Pi \vdash \textsc{PWF}^T_\mu(\lceil c \rceil \overline{\widehat{r} X})}\ \textsc{PWF-FC}$$

Lemma 5.49.

$$\frac{\Gamma;\Pi \vdash PWF^{F' \overline{\widehat{r} X'}}_\mu\left(F \overline{\widehat{r} X}\right) \qquad \Gamma;\Pi \vdash F \overline{\widehat{r} X} \parallel_\mu F' \overline{\widehat{r} X'} \qquad \Gamma \vdash F' \overline{\widehat{r} X'} :_t \tau}{\forall i. \qquad \Gamma \vdash |X_i|[\mu] \approx X'_i}$$

Proof. If $F = \lfloor f \rfloor$, we obtain the required conversion from Definition 5.13; if $F = c$ or $F = \lceil c \rceil$ where c is a constructor, we obtain the conversion from Lemma 5.47. □

Lemma 5.50 (Pattern lemma)**.** *Pattern matching produces well-typed substitutions.*

$$\frac{\vdash \textsc{Defs}(\Gamma) \qquad \Gamma \vdash \textsc{Defs}(\Pi) \qquad \textsc{Dom}(\mu) \subseteq \mathrm{BV}(\Pi) \qquad \Gamma;\Pi \vdash PWF^T_\mu(P) \qquad \Gamma;\Pi \vdash P :^q_r \lambda \qquad \Gamma;\Pi \vdash P \parallel_\mu T \qquad \Gamma \vdash T :_r \tau}{\Gamma \vdash \mu :^{FPV_\Pi(P)} \Pi \qquad \Gamma \vdash |P|[\mu] :_r \tau}$$

Proof. By induction on the structure of the pattern, unwrapping the proof of pattern match and well-formedness, and inverting the typing judgements using Lemma 5.35.

By well-formedness of the pattern (PWF implies either $\textsc{PatWF}_F$ or $\textsc{PatWF}_f$), P is an (n-ary, $n \geq 0$) application headed by $\lfloor f \rfloor$ or by a (possibly forced) constructor c. Let us call this head h and proceed by (nested) induction on the size of this application: two base cases followed by the inductive step.

If $P = h = c$ **where** $(c :_s \sigma = \textbf{constructor} \in \Gamma)$**,** then $\mathrm{FPV}_\Pi(P) = \emptyset$ by disjointness of binders and well-typedness of μ holds trivially.

By MatchCtor, $T = c$. Since $|P|[\mu] = c$ as well – names bound in Γ are not in $\textsc{Dom}(\mu)$ – we obtain $\Gamma \vdash |P|[\mu] :_r \tau$ by assumption.

If $P = h = \lceil c \rceil$ **where** $(c :_s \sigma = \textbf{constructor} \in \Gamma)$**,** we proceed like in the previous case, except that instead of inverting MatchCtor, we use Lemma 5.27.

If $P = h = \lfloor f \rfloor$ **where** $(f :_s \phi = \overline{C} \in \Gamma)$, then $\mathrm{FPV}_\Pi(P) = \emptyset$ by disjointness of binders and well-typedness of μ holds trivially.

By MatchDefName, $T = f$. Since $|P|[\mu] = f$ as well – names bound in Γ are not in $\mathrm{Dom}(\mu)$ – we obtain $\Gamma \vdash |P|[\mu] :_r \tau$ by assumption.

If $P = F \mathrel{\widehat{s}} X$ **where** $F = h \mathrel{\widehat{\overline{u}}} \overline{Y}$, then by pattern match, we have $T = F' \mathrel{\widehat{s'}} X'$ where $F' = |h| \mathrel{\widehat{\overline{u'}}} \overline{Y'}$ such that $\Gamma; \Pi \vdash F \parallel_\mu F'$ and $\Gamma; \Pi \vdash X \parallel_\mu X'$. By InvPatApp, we have $\Gamma; \Pi \vdash F :^q_r (x :_s \sigma) \to \rho$ and $\Gamma; \Pi \vdash X :^q_{r \wedge s} \sigma$ and $\Gamma, \Pi \vdash \tau \approx \rho[x \mapsto X]$ for some name x and types σ and ρ. By InvApp, we have $\Gamma \vdash F' :_r (x' :_{s'} \sigma') \to \rho'$ and $\Gamma \vdash X' :_{r \wedge s'} \sigma'$ and $\Gamma \vdash \tau \approx \rho'[x' \mapsto X']$ for some name x' and types σ' and ρ'. We verify that the requirements for induction are satisfied and thus obtain $\Gamma \vdash \mu :^{\mathrm{FPV}_\Pi(F)} \Pi$, and $\Gamma \vdash |F|[\mu] :_r (x' :_{s'} \sigma') \to \rho'$, and $\Gamma \vdash (x' :_{s'} \sigma') \to \rho' \approx \big((x :_s \sigma) \to \rho\big)[\mu]$. By disjointness of binders, $\Gamma \vdash (x' :_{s'} \sigma') \to \rho' \approx (x :_s \sigma[\mu]) \to \rho[\mu]$. By Lemma 5.40, we have $x = x'$, $s = s'$, $\Gamma \vdash \sigma' \approx \sigma[\mu]$, and $\Gamma, (x :_s \sigma') \vdash \rho' \approx \rho[\mu]$. In the following, We will therefore write x and s instead of x' and s'.

To finish the proof, we case-split on X. From well-formedness of P, we have $\Gamma; \Pi \vdash \mathrm{PatWF}_X(X)$ in both possible cases $h = c$ and $h = \lfloor f \rfloor$.

- **If** $X = c$ **where** $(c :_v \varsigma = \textbf{constructor}) \in \Gamma$, then $\mathrm{FPV}_\Pi(F) = \mathrm{FPV}_\Pi(P)$ and well-typedness of μ on $\mathrm{FPV}_\Pi(P)$ follows directly from the inductive hypothesis. By inversion of MatchCtor, we have $X' = c = |X|[\mu]$. Since $\Gamma \vdash X' :_{r \wedge s} \sigma'$, we have $\Gamma \vdash |X|[\mu] :_{r \wedge s} \sigma'$. We can apply App to obtain $\Gamma \vdash |F \mathrel{\widehat{s}} X|[\mu] :_r \rho'[x \mapsto |X|[\mu]]$. Since $|X|[\mu] = X'$ and $\Gamma \vdash \tau \approx \rho'[x \mapsto X']$, we can use Conv to conclude $\Gamma \vdash |F \mathrel{\widehat{s}} X|[\mu] :_r \tau$, as required.
- **If** $X = \lceil c \rceil$ **where** $(c :_v \varsigma = \textbf{constructor}) \in \Gamma$, we proceed like in the previous case, except that instead of inverting MatchCtor, we use Lemma 5.27.
- **If** $X = n$ **where** $(n :_v \varsigma) \in \Pi$, then from MatchPatVar, $\mu(n) = X'$ and by InvPatRef, $v = r \wedge s$ and $\Gamma, \Pi \vdash \varsigma \approx \sigma$. We already have $\Gamma \vdash X' :_{r \wedge s} \sigma'$ and $\Gamma \vdash \sigma' \approx \sigma[\mu]$, which yields $\Gamma \vdash \mu(n) :_v \sigma[\mu]$ by Conv. We thus obtain $\Gamma \vdash \mu(n) :_v \sigma[\mu]$, and by Lemma 5.31 and Conv, $\Gamma \vdash \mu(n) :_v \varsigma[\mu]$.
  Since $|X|[\mu] = \mu(n) = X'$, we can just use the earlier result $\Gamma \vdash X' :_{r \wedge s} \sigma'$ and (part of) the inductive hypothesis $\Gamma \vdash |F|[\mu] :_r (x :_s \sigma') \to \rho'$ together with App to obtain $\Gamma \vdash |F \mathrel{\widehat{s}} X|[\mu] :_r \rho'[x \mapsto X']$. Finally, by Conv, $\Gamma \vdash |F \mathrel{\widehat{s}} X|[\mu] :_r \tau$.
- **If** $X = G \mathrel{\widehat{v}} Z$, then $\Gamma; \Pi \vdash \mathrm{PatWF}_X(X)$ is equivalent to $\Gamma; \Pi \vdash \mathrm{PatWF}_F(X)$. By well-formedness of X, pattern G must be (possibly forced-) constructor-headed, and by Lemma 5.49, we have $\Gamma \vdash |X|[\mu] \approx X'$, and thus we can use PWF-c or PWF-fc to obtain $\Gamma; \Pi \vdash \mathrm{PWF}^{X'}_\mu(X)$. We can therefore apply (top-level) induction to X to obtain $\Gamma \vdash \mu :^{\mathrm{FPV}_\Pi(X)} \Pi$, which, together with the earlier result $\Gamma \vdash \mu :^{\mathrm{FPV}_\Pi(F)} \Pi$, yields $\Gamma \vdash \mu :^{\mathrm{FPV}_\Pi(P)} \Pi$.
  Furthermore, from induction we also have $\Gamma \vdash |X|[\mu] :_{r \wedge s} \sigma'$, and we can therefore use App (and Conv) to conclude $\Gamma \vdash |F \mathrel{\widehat{s}} X|[\mu] :_r \tau$.

If $X = [T]$, then $\mathrm{FPV}_\Pi(F) = \mathrm{FPV}_\Pi(P)$ and well-typedness of μ on $\mathrm{FPV}_\Pi(P)$ follows directly from the inductive hypothesis.

By InvPatForced, $\Gamma, \Pi \vdash T :_{r\wedge s} \sigma$. By Lemma 5.34, $\Gamma \vdash T[\mu] :_{r\wedge s} \sigma[\mu]$ and by our earlier result $\Gamma \vdash \sigma' \approx \sigma[\mu]$, we have $\Gamma \vdash T[\mu] :_{r\wedge s} \sigma'$. We can rewrite this using $T = |X|$ and use App and our earlier result $\Gamma \vdash |F|[\mu] :_r (x :_s \sigma') \to \rho'$ to obtain $\Gamma \vdash |F \mathbin{\widehat{s}} X|[\mu] :_r \rho'[x \mapsto |X|[\mu]]$. By Lemma 5.49, we have $\Gamma \vdash |X|[\mu] \approx X'$. By Lemma 5.38, we have $\Gamma \vdash \rho'[x \mapsto |X|[\mu]] \approx \rho'[x \mapsto X']$, and by Conv we can conclude $\Gamma \vdash |F \mathbin{\widehat{s}} X|[\mu] :_r \rho'[x \mapsto X']$, as required.

If $X = \lfloor f \rfloor$, we have a contradiction with $\Gamma; \Pi \vdash \mathrm{PatWF}_X(X)$ obtained from the property PWF.

□

Lemma 5.51 (Pattern lemma: conversion)**.**

$$\frac{\vdash \mathit{Defs}(\Gamma) \qquad \Gamma \vdash \mathit{Defs}(\Pi) \qquad \mathit{FPV}_\Pi(P) = \mathit{Dom}(\mu) = \mathrm{BV}(\Pi) \qquad \Gamma;\Pi \vdash \mathit{PWF}^T_\mu(P) \qquad \Gamma;\Pi \vdash P :^q_r \lambda \qquad \Gamma;\Pi \vdash P \parallel_\mu T \qquad \Gamma \vdash T :_r \tau}{\Gamma \vdash \tau \approx \lambda[\mu]}$$

Proof. By Lemma 5.50, we have $\Gamma \vdash \mu : \Pi$ and $\Gamma \vdash |P|[\mu] :_r \tau$. We apply Lemma 5.48 to obtain $\Gamma, \Pi \vdash |P| :_r \lambda$, Lemma 5.34 to obtain $\Gamma \vdash |P|[\mu] :_r \lambda[\mu]$, and Lemma 5.41 to obtain $\Gamma \vdash \tau \approx \lambda[\mu]$. □

5.7.9 Subject reduction

Theorem 5.1 (Subject reduction)**.** *Let M and N be terms of $TT^{RE}_\star$ and let Γ be a well formed environment. Then reduction preserves types, as expressed by the following rule.*

$$\frac{\Gamma \vdash M \rightsquigarrow N \qquad \Gamma \vdash M :_r \tau}{\Gamma \vdash N :_r \tau}\ \mathit{SubjectReduction}$$

Proof. By induction on the length of the reduction sequence, according to possible reduction steps listed in Section 5.4. In the following proof, whenever we invoke induction with an augmented environment, we can observe that the addition is always well typed, and therefore the environment stays well formed.

RedVar with $M = n$ for a name n.

From RedVar, we know that $(n :_r \tau' = N) \in \Gamma$ for some τ'. Because Γ is well formed, we have $\Gamma', (n :_r \tau' = N) \sqsubseteq \Gamma$ for some Γ' such that $\Gamma', n :_r \tau' \vdash N :_r \tau'$. By Thinning (Lemma 5.4), we obtain $\Gamma \vdash N :_r \tau'$. By InvRef, we have $\Gamma \vdash \tau' \approx \tau$, and by Conv, $\Gamma \vdash N :_r \tau$.

Redex with $M = (\lambda x :_s \sigma.\ T) \mathbin{\widehat{s}} X$, $N = T[x \mapsto X]$.

By InvApp, $\Gamma \vdash \tau \approx \rho[x \mapsto X]$ for some x and ρ. We also have $\Gamma \vdash (\lambda x :_s \sigma.\ T) : (x :_s \sigma) \to \rho$ and $\Gamma \vdash X :_{r\wedge s} \sigma$ for some σ.

Further, by InvLam, $\Gamma, x :_{s'} \sigma' \vdash T :_r \rho'$ such that $\Gamma \vdash (x :_{s'} \sigma') \to \rho' \approx (x :_s \sigma) \to \rho$. By Lemma 5.40, we have $s = s'$ and $\Gamma \vdash \sigma \approx \sigma'$, and $\Gamma, x :_s \sigma \vdash \rho \approx \rho'$.

By Lemma 5.32, we have $\Gamma \vdash T[x \mapsto X] :_r \rho'[x \mapsto X]$. By Lemma 5.31, we have $\Gamma \vdash T[x \mapsto X] :_r \rho[x \mapsto X]$, and by Conv, $\Gamma \vdash T[x \mapsto X] :_r \tau$.

RedClauses with $M = f\ \overline{\widehat{s'} X'}$, $N = R_k[\mu]$ for some R_k and μ.

By inversion of RedClauses, we have $\Gamma; \Pi_k \vdash L_k \parallel_\mu f\ \overline{\widehat{s'} X'}$, as well as $\Gamma; \Pi_k \vdash \textsc{PatWF}_f(L_k)$, also $(f :_t \phi = \overline{C}) \in \Gamma$, and $\mathrm{FPV}_{\Pi_k}(L_k) = \textsc{Dom}(\mu)$. By well-formedness of Γ, especially from Clause for C_k of f, we have $\mathrm{FPV}_{\Pi_k}(L_k) = \mathrm{BV}(\Pi)$, and also $L_k = \lfloor f \rfloor \overline{\widehat{s} X}$, such that $\Gamma; \Pi_k \vdash L_k :^t_t \rho$, where $\Gamma, \Pi_k \vdash R_k :_t \rho$, as well.

Using PWF-f, we have $\Gamma; \Pi_k \vdash \mathrm{PWF}^M_\mu(L_k)$, and we can use Lemma 5.51 to obtain $\Gamma \vdash \tau \approx \rho[\mu]$. By Lemma 5.50, $\Gamma \vdash \mu : \Pi_k$. By Lemma 5.34, $\Gamma \vdash R_k[\mu] :_t \rho[\mu]$, and by Conv, $\Gamma \vdash R_k[\mu] :_t \tau$.

Finally, by repeated InvApp and InvRef applied to the judgement $\Gamma \vdash f\ \overline{\widehat{s'} X'} :_r \tau$, we know that f is bound with retention t such that $r \leqslant t$. Then we can use Lemma 5.11 to conclude $\Gamma \vdash R_k[\mu] :_r \tau$, as required.

RedLetElim with $M = \textbf{let}\ d\ \textbf{in}\ N$ and $d = (n :_s \sigma = b)$.

From RedLetElim, we have $n \notin \mathrm{FV}(M)$. By InvLet, we have $\Gamma, d \vdash N :_r \tau$. By Lemma 5.30, we get $\Gamma \vdash N :_r \tau$.

RedLetAppL with $M = (\textbf{let}\ d\ \textbf{in}\ F)\ \widehat{s}\ X$, and $N = (\textbf{let}\ d\ \textbf{in}\ F\ \widehat{s}\ X)$, and the definition $d = (n :_s \sigma = b)$.

By InvApp, we have $\Gamma \vdash \tau \approx \rho[n \mapsto X]$ for some type ρ. We also have $\Gamma \vdash (\textbf{let}\ d\ \textbf{in}\ F) : (n :_s \sigma) \to \rho$ and $\Gamma \vdash X :_{r \wedge s} \sigma$ for some type σ.

By InvLet, we have $\Gamma, d \vdash F : (n :_s \sigma) \to \rho$ and $\Gamma \vdash \textsc{Def}(d)$.

By Lemma 5.3, $\Gamma, d \vdash X :_{r \wedge s} \sigma$, and thus we can use App to obtain $\Gamma, d \vdash F\ \widehat{s}\ X :_r \rho[n \mapsto X]$. Finally, we use Let and Conv to conclude $\Gamma \vdash \textbf{let}\ d\ \textbf{in}\ (F\ \widehat{s}\ X) :_r \tau$.

RedAppR with $M = F\ \widehat{s}\ X$ and $N = F\ \widehat{s}\ X'$, where $\Gamma \vdash X \rightsquigarrow X'$

We use InvApp, induction, and App again to obtain $\Gamma \vdash \tau \approx \rho[x \mapsto X]$ and $\Gamma \vdash N :_r \rho[x \mapsto X']$. By ConvRed, we have $\Gamma \vdash X \approx X'$ and by Lemma 5.14 together with Conv, we have $\Gamma \vdash N :_r \rho[x \mapsto X]$, as required.

All remaining rules are purely structural, solved by induction. They follow the same structure: look at the last rule in the derivation, apply induction to its premises, observe that the prerequisites for the re-application of the rule are satisfied (the types have not changed), and finally re-apply the rule. □

5.7.10 Correctness

5.7.10.1 Erasure respects pattern matching

This section explains that erasure preserves pattern matching. In order to select the correct pattern clause for reduction after erasure, we must ensure that:

- the left hand side of the correct clause matches the given term;
- the preceding clauses do *not* match the given term.

We will use these results to prove commutativity of erasure and reduction for the remaining elements of the whole calculus.

Lemma 5.52. *Erasure preserves linearity of patterns.*

Proof. Erasure only deletes pattern variables and cannot create new ones. □

Lemma 5.53 (Pattern matching is preserved by erasure)**.** *Well-typed pattern matching in runtime contexts is preserved by erasure.*

$$\frac{\Gamma;\Pi \vdash P :^{q}_{R} \lambda \qquad \Gamma;\Pi \vdash P \parallel_{\mu} T \qquad \Gamma \vdash T :_{R} \tau}{\langle\Gamma\rangle;\langle\Pi\rangle \vdash \langle P\rangle \parallel_{\langle\mu\rangle/\langle\Pi\rangle} \langle T\rangle}$$

Proof. We proceed by induction on the structure of P.

If $P = c$ **where** $(c :_s \sigma = \textbf{constructor}) \in \Gamma$**,** then $\langle P\rangle = c$ and $\langle T\rangle = c$. By InvRef, c is R-bound in Γ and therefore the binding survives erasure and by MatchCtor, the match holds after erasure.

If $P = n$ **where** $(n :_s \sigma) \in \Pi$**,** then $\mu(n) = T$ and $\langle n\rangle = n$. By InvRef, n is R-bound in Π and therefore the binding survives erasure and $n \in \mathrm{Dom}(\langle\mu\rangle/\langle\Pi\rangle)$. We apply MatchPatVar.

If $P = \lfloor f \rfloor$ **where** $(f :_s \sigma = \overline{C}) \in \Gamma$**,** then $\langle P\rangle = \lfloor f \rfloor$ and $\langle T\rangle = f$. By InvRef, f is R-bound in Γ and therefore the binding survives erasure. We apply MatchDefName.

If $P = \lceil c \rceil$ **where** $(c :_s \sigma = \textbf{constructor}) \in \Gamma$**,** then $\langle P\rangle = \lceil c \rceil$ and $\langle T\rangle = c$. By InvPatForcedCtor, c is R-bound, its binding survives erasure, and we can apply MatchForcedCtor.

If $P = [M]$**,** then $\langle P\rangle = [\langle M\rangle]$ and we can use MatchForced.

If $P = F \mathbin{\widehat{s}} X$**,** then by inversion of MatchApp, $T = F' \mathbin{\widehat{s'}} X'$ such that $\Gamma;\Pi \vdash F \parallel_{\mu} F'$ and $\Gamma;\Pi \vdash X \parallel_{\mu} X'$. By InvPatApp, we have $\Gamma;\Pi \vdash F :^{q}_{\mathsf{R}} (x :_s \sigma) \to \rho$ and $\Gamma;\Pi \vdash X :^{q}_{s} \sigma$ such that $\Gamma \vdash \lambda \approx \rho[x \mapsto |X|]$. By InvApp, we have $\Gamma \vdash F' :_{\mathsf{R}} (x' :_{s'} \sigma) \to \rho'$ and $\Gamma \vdash X' :_{s'} \sigma$ such that $\Gamma \vdash \tau \approx \rho'[x' \mapsto X']$.

Since $\Gamma;\Pi \vdash F \parallel_{\mu} F'$ and both F and F' are well-typed in R-contexts, we obtain $\langle\Gamma\rangle;\langle\Pi\rangle \vdash \langle F\rangle \parallel_{\langle\mu\rangle/\langle\Pi\rangle} \langle F'\rangle$ by induction.

- **If** $s = \mathbf{R}$**,** then X and X' are well-typed in R as well and we can use induction to obtain $\langle\Gamma\rangle;\langle\Pi\rangle \vdash \langle X\rangle \parallel_{\langle\mu\rangle/\langle\Pi\rangle} \langle X'\rangle$ and finally MatchApp.
- **If** $s = \mathbf{E}$**,** then $\langle P\rangle = \langle F\rangle$ and $\langle T\rangle = \langle F'\rangle$, and match between them is proven already by the inductive hypothesis.

□

Lemma 5.54 (Pattern mismatches always appear in runtime contexts)**.**

$$\frac{\Gamma;\Pi \vdash P :^{q}_{r} \tau \qquad \Gamma \vdash P \not\parallel T}{q \leqslant r}$$

Therefore, if the whole function definition survives erasure, i.e. $q = \mathsf{R}$, then $r = \mathsf{R}$.

Proof. By induction on the derivation of mismatch.

MismatchHead with $P = c\,\overline{\widehat{r} X}$. By iteration of InvPatApp and InvPatRef, we obtain $q \leqslant r$.

MismatchArg or MismatchArgForced or MismatchLHS $P = h\,\overline{\widehat{r} X}$, and $T = |h|\,\overline{\widehat{r'} X'}$, where h is a (possibly forced) constructor or $\lfloor f \rfloor$. By inversion of the mismatch rule, we have $\Gamma \vdash X_i \not\parallel X'_i$ for some i. By iteration of InvPatApp, $\Gamma;\Pi \vdash X_i :^{q}_{r \wedge s} \sigma$ for some s and σ. We can apply induction to X_i and X'_i to obtain $q \leqslant r \wedge s$. Since $r \wedge s \leqslant r$, we obtain $q \leqslant r$, as required.

□

Corollary 5.55. *It is not possible to obtain a mismatch without runtime inspection.*

Lemma 5.56 (Pattern mismatches are preserved by erasure)**.** *Let $\Gamma;\Pi$ be a well formed pair of environments. Then the following holds.*

$$\frac{\Gamma;\Pi \vdash P :^{R}_{r} \lambda \qquad \Gamma,\Pi \vdash T :_{r} \tau \qquad \Gamma \vdash P \not\parallel T}{\langle\Gamma\rangle \vdash \langle P\rangle \not\parallel \langle T\rangle}$$

Proof. Informally, the derivation $\Gamma \vdash P \not\parallel T$ has a linear shape – invocations of the rule MismatchArg(Forced) define a path through the tree of nested applications that leads to a mismatch in constructors marked by rule MismatchHead. By pattern rule PatCtor, MismatchHead points at a subterm in a runtime/retained context (thanks to the requirement $q \leqslant r$) and by PatApp, all supercontexts are retained at runtime, too. This means that the whole path through the tree of applications survives erasure and causes a mismatch in the erased term.

Formally, we proceed by induction on the derivation $\Gamma \vdash P \not\parallel T$.

MismatchHead with $P = c\,\overline{\widehat{r} X}$ and $T = c'\,\overline{\widehat{r'} X'}$ where $c \neq c'$ are both constructors in Γ. By iteration of InvPatApp and InvPatRef, $r = \mathsf{R}$ and $c \in \mathrm{RBV}(\Gamma)$. By iteration of InvApp and InvRef, $c' \in \mathrm{RBV}(\Gamma)$. Since both $c, c' \in \mathrm{RBV}(\Gamma)$, they survive erasure as constructors in $\langle\Gamma\rangle$. By the definition of erasure, $\langle P\rangle$ is a (potentially nullary) pattern application headed by c and $\langle T\rangle$ is a (potentially nullary) application headed by c' with $c \neq c'$. Therefore, MismatchHead is applicable after erasure.

MismatchArg with $P = c\,\widehat{r}\,\overline{X}$ for some F and $T = c\,\widehat{r}\,\overline{X'}$, where $\exists i.\ \Gamma \vdash X_i \not\parallel X'_i$ and $(c :_t \phi = \textbf{constructor}) \in \Gamma$.

By iteration of InvPatApp, we have $\Gamma; \Pi \vdash X_i :^{\mathsf{R}}_{r \wedge s} \sigma$ for some s and σ. Since both P and T are headed by the same constructor, by iteration of InvApp we get $\Gamma \vdash X'_i :_{r \wedge s} \sigma$, as well. However, by Lemma 5.54, we then have $r \wedge s = \mathsf{R}$, and thus the i-th arguments of both P and T survive erasure. By iteration of InvApp and InvRef, we have $r \leqslant t$, and since $r \wedge s = \mathsf{R}$, we have $r = \mathsf{R}$ and $t = \mathsf{R}$. The binding of c therefore survives erasure, too and we can use MismatchArg to derive mismatch after erasure.

MismatchLHS Like MismatchArg.

MismatchArgForced Like MismatchArg.

□

5.7.10.2 Erasure respects reduction

Theorem 5.2 (Correctness of erasure)**.** *Let Γ be a well formed environment, let T and T' be terms of $TT^{RE}_{\star}$, and let T be well typed in runtime contexts. Then erasure commutes with reduction.*

$$\frac{\Gamma \vdash T :_R \tau \qquad \Gamma \vdash T \rightsquigarrow T'}{\textit{either}\ \langle T \rangle = \langle T' \rangle\ \textit{or}\ \langle \Gamma \rangle \vdash \langle T \rangle \rightsquigarrow \langle T' \rangle}$$

NB.: The proof of this theorem does not depend on the Church-Rosser property.

Proof. By induction on the derivation of reduction.

RedVar where $T = n$ and $(n :_r \tau = T') \in \Gamma$. By InvRef, we have $\mathsf{R} \leqslant r$, which means that $r = \mathsf{R}$. By the definition of erasure on environments, we then have $(n = \langle T' \rangle) \in \langle \Gamma \rangle$, and therefore, by RedVar, $\langle \Gamma \rangle \vdash n \rightsquigarrow \langle T' \rangle$.

Redex where $T = (\lambda x :_s \sigma.\ M)\,\widehat{t}\,X$ and $T' = M[x \mapsto X]$. By InvApp and InvLam, we have $s = t$, and because $\mathsf{R} \wedge s = s$, we also have $\Gamma, x :_s \sigma \vdash M :_{\mathsf{R}} \rho$ for some ρ and $\Gamma \vdash X :_s \sigma$.

If $s = \mathsf{R}$, then by the definition of erasure, $\langle T \rangle = (\lambda x.\ \langle M \rangle)\ \langle X \rangle$, and $\langle T' \rangle = \langle M[x \mapsto X] \rangle$. By Redex and the definition of erasure,

$$\langle \Gamma \rangle \vdash (\lambda x.\ \langle M \rangle)\ \langle X \rangle \rightsquigarrow \langle M \rangle[x \mapsto \langle X \rangle]$$

By Lemma 5.42, $\langle T' \rangle = \langle M \rangle[x \mapsto \langle X \rangle]$, and therefore $\langle \Gamma \rangle \vdash \langle T \rangle \rightsquigarrow \langle T' \rangle$.

If $s = \mathsf{E}$, then $\langle T \rangle = \langle M \rangle$. By Variable Survival (Lemma 5.46), $x \notin \mathrm{FV}(\langle M \rangle)$, and therefore $\langle T' \rangle = \langle M[x \mapsto X] \rangle = \langle M \rangle[x \mapsto \langle X \rangle] = \langle M \rangle = \langle T \rangle$.

RedClauses with $T = f\overline{\widehat{r}X}$ and $T' = R_k[\mu]$. We will show that the reduction rule is applicable after erasure, and therefore we need to show that its prerequisites hold after erasure.

By iteration of InvApp, we obtain $\Gamma \vdash f :_{\mathsf{R}} \overline{(x :_s \sigma)} \to \rho$ and $\Gamma \vdash X_i :_{s_i} \sigma_i$ for each i.

Since, by InvRef, f is R-bound in Γ, the binding of f survives erasure. This is related to the first requirement of RedClauses.

The second requirement of RedClauses is not affected by erasure, as it just defines names of components of pattern matching clauses.

Since f is R-bound, and Γ is well-formed, we have $\Gamma; \Pi_k \vdash L_k :^{\mathsf{R}}_{\mathsf{R}} \lambda_k$ for some λ_k, and by Lemma 5.45, we have $\langle\Gamma\rangle; \langle\Pi_k\rangle \vdash \textsc{PatWF}_f(\langle L_k\rangle)$. This is the third requirement of RedClauses.

Furthermore, all clauses C_i for $i < k$ still mismatch after erasure, by Lemma 5.56, which means that reduction after erasure does not get stuck before it gets to the matching clause. This is the fourth requirement of RedClauses.

By well-formedness of Γ and well-typedness of T, Lemma 5.53 yields $\langle\Gamma\rangle; \langle\Pi\rangle \vdash \langle L_k\rangle \parallel_{\langle\mu\rangle/\langle\Pi\rangle} \langle T\rangle$. This is the fifth requirement of RedClauses.

Since $\textsc{Dom}(\mu) = \mathrm{BV}(\Pi)$, we have $\textsc{Dom}(\langle\mu\rangle/\langle\Pi\rangle) = \mathrm{BV}(\langle\Pi\rangle)$. This is the last requirement of RedClauses.

Therefore, we can apply RedClauses after erasure, too, to obtain $\langle\Gamma\rangle \vdash \langle f\overline{\widehat{r}X}\rangle \rightsquigarrow \langle R_k\rangle[\langle\mu\rangle/\langle\Pi\rangle]$. Since from well-formedness of Γ (and Lemma 5.5), we have $\Gamma, \Pi \vdash R_k :_{\mathsf{R}} \lambda_k$, by Lemma 5.46, free variables of $\langle R_k\rangle$ are R-bound in either Γ or Π, and thus $\mathrm{FV}(\langle R_k\rangle) \cap (\textsc{Dom}(\mu) \setminus \mathrm{BV}(\langle\Pi\rangle)) = \emptyset$. Therefore $\langle R_k\rangle[\langle\mu\rangle/\langle\Pi\rangle] = \langle R_k\rangle[\langle\mu\rangle]$, since the names added to the substitution do not occur in $\langle R_k\rangle$, anyway. Hence $\langle\Gamma\rangle \vdash \langle f\overline{\widehat{r}X}\rangle \rightsquigarrow \langle R_k\rangle[\langle\mu\rangle]$ and by Lemma 5.43, $\langle\Gamma\rangle \vdash \langle f\overline{\widehat{r}X}\rangle \rightsquigarrow \langle R_k[\mu]\rangle$.

RedLetElim with $T = \textbf{let}\ n :_s \sigma = b\ \textbf{in}\ M$ and $T' = M$, where $n \notin \mathrm{FV}(M)$.

If $s = \textbf{E}$, then $\langle T\rangle = \langle M\rangle = \langle T'\rangle$.

If $s = \textbf{R}$, then clearly $n \notin \mathrm{FV}(\langle M\rangle)$ and we can apply RedLetElim after erasure.

RedLetAppL with $T = (\textbf{let}\ n :_s \sigma = b\ \textbf{in}\ M)\,\widehat{r}\,X$ and $T' = \textbf{let}\ n :_s \sigma = b\ \textbf{in}\ (M\,\widehat{r}\,X)$.

If $s = \textbf{E}$, we can observe, combining several definitional rules of erasure, that $\langle T\rangle = \langle M\,\widehat{r}\,X\rangle = \langle T'\rangle$.

If $s = \textbf{R}$, then RedLetApp is applicable to $\langle T\rangle$, yielding $\langle T'\rangle$.

Structural rules The remaining reduction rules are purely structural and are solved by induction on the derivation of $\Gamma \vdash T \rightsquigarrow T'$.

- If reduction occurs in any (direct) subterm that itself occurs in a runtime context, the subterm can be passed to the induction hypothesis and the reduction rule can be re-applied after erasure.

- If reduction occurs in any (direct) subterm that itself occurs in an erased context, such as types, the subterm will be erased (or replaced by □), yielding $\langle T \rangle = \langle T' \rangle$ trivially.

□

Remark 5.9. Mishra-Linger remarks [ML08] that the computation/work saved by erasure corresponds exactly to the cases where $\langle T \rangle = \langle T' \rangle$ in Theorem 5.2. This means that while the unerased program needed to perform a step of computation (reduction), the erased program saves it.

Corollary 5.57. *Let Γ be a well formed environment, let T and T' be terms of $TT^{RE}_{\star}$, and let T be well typed in runtime contexts. Then erasure commutes with repeated reduction.*

$$\frac{\Gamma \vdash T :_R \tau \qquad \Gamma \vdash T \rightsquigarrow^\star T'}{\langle \Gamma \rangle \vdash \langle T \rangle \rightsquigarrow^\star \langle T' \rangle}$$

Proof. By induction on the length of the reduction sequence.

If $T = T'$, then $\langle T \rangle = \langle T' \rangle$ and by reflexivity of $\rightsquigarrow^\star$, we have $\langle \Gamma \rangle \vdash \langle T \rangle \rightsquigarrow^\star \langle T' \rangle$.

If $T \neq T'$, there is a term N such that $\Gamma \vdash T \rightsquigarrow N \rightsquigarrow^\star T'$. By Theorem 5.2, $\langle \Gamma \rangle \vdash \langle T \rangle \rightsquigarrow \langle N \rangle$ or $\langle T \rangle = \langle N \rangle$. By Subject Reduction (Theorem 5.1), we have $\Gamma \vdash N :_R \tau$ and therefore we can invoke induction to obtain $\langle \Gamma \rangle \vdash \langle N \rangle \rightsquigarrow^\star \langle T' \rangle$. Finally, by transitivity of of $\rightsquigarrow^\star$, we obtain $\langle \Gamma \rangle \vdash \langle T \rangle \rightsquigarrow^\star \langle T' \rangle$. □

Corollary 5.58. *If a type-correct $TT^{RE}_{\star}$ program P reduces to a value X which is* primitive *(it erases to itself, such as a nullary constructor), then $\langle P \rangle$ reduces to the same value.*

Proof. From Corollary 5.57. □

Corollary 5.59 (Erasure preserves conversion)**.**

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

Proof. By Church-Rosser (Lemma 5.28), T and T' have a common reduct. By Corollary 5.57, this is preserved by erasure. □

5.7.11 Future work

The main item of future work is proving that reduction in $\mathsf{TT}_\star$ is confluent (Conjecture 5.1).

Conjecture 5.2 (Reverse simulation)**.** *Let Γ be a well formed environment, let T be a term of $TT^{RE}_{\star}$, and let T be well typed in runtime contexts. Then if $\langle T \rangle$ reduces in $\langle \Gamma \rangle$, there is an*

equivalent reduction sequence in the unerased term.

$$\frac{\Gamma \vdash T :_R \tau \qquad \langle\Gamma\rangle \vdash \langle T\rangle \rightsquigarrow^\star T''}{\exists T'.\ T'' = \langle T'\rangle \wedge \left(\Gamma \vdash T \rightsquigarrow^\star T'\right)}$$

This means that erased programs cannot reduce in ways that unerased programs would not. It would also imply preservation of strong normalisation by erasure.

Remark 5.10. $\mathsf{TT}_\star$, as defined in this chapter, is not strongly normalising, since it allows general recursion. A more restricted reduction behaviour can be achieved by adding a separate termination checker.

Progress We already have Preservation (Theorem 5.1) and it would be useful to define values and coverage of patterns and show Progress, as well.

Forced patterns I do not discuss forced patterns formally beyond their characterisation in Definition 5.13 (and a rather informal connection between forced constructors and single-branch case trees in Section 7.2.3.2). For implementations, it will be necessary to define a formal procedure to check consistency of forced patterns in $\mathsf{TT}_\star$.

Semantics In this dissertation, I do not discuss the denotational semantics of $\mathsf{TT}_\star$ at all. It would be useful to find a model for $\mathsf{TT}_\star$ and prove the soundness of the typing rules with respect to it.

Chapter 6

Erasure inference

$\mathsf{TT}_\star$, being a core language, needs to be fully annotated with explicit erasure annotations for each binder and application (besides explicit type annotations). It would be unacceptable to require programmers to provide so much erasure annotation, and therefore we need a way to infer it from a given program.

Indeed, experience with the implementation of erasure presented in Chapter 4, currently in the Idris compiler, and the implementation of erasure presented in Chapter 5 together with inference presented in this chapter, implemented in a small toy compiler, suggests that we can infer annotations from a completely erasure-unannotated program. Erasure inference therefore removes the need to clutter programs with explicit annotations and the whole erasure pipeline can take place entirely[1] behind the scenes.

This chapter presents an algorithm for erasure inference and shows its soundness and completeness with respect to the checking rules from Chapter 5, along with its optimality in a certain sense.

6.1 Overview

Figure 5.5 shows the erasure pipeline, together with the corresponding variants of $\mathsf{TT}_\star$ at each stage.

In the figure, erasure inference covers the transition from $\mathsf{TT}^\bullet_\star$ to $\mathsf{TT}^{\mathsf{RE}}_\star$. First, we number all unknown erasure annotations to obtain a program in $\mathsf{TT}^{\mathsf{evar}}_\star$. Then we typecheck the program using a specialised set of typing rules (Section 6.3) that generate erasure constraints. We solve the set of erasure constraints (Section 6.4) to obtain the minimal set of variable bindings (and applications) that have to be marked as retained. These bindings and applications are annotated with R and all remaining annotations are set to E. This produces a fully annotated program in $\mathsf{TT}^{\mathsf{RE}}_\star$, which we can check using the rules given in Chapter 5.

6.1.1 Erasure variables

Erasure variables, shortly *evars*, occur in $\mathsf{TT}^{\mathsf{evar}}_\star$ programs and they represent erasure annotations that have not been set to a specific value (R or E) by the programmer.

[1] With the exception of declarations of compiler built-ins, FFI calls, etc.; see Section 7.5.

Evars are numbered (rather than given single-letter names) and each of them stands for a particular place in the program where we still need to fill an erasure annotation. Where unambiguous, I will identify evars with their numbers.

6.1.2 Structure of erasure annotations

Each evar will eventually be assigned a definite value – either E or R. I arrange these two values in a lattice $\mathsf{E} < \mathsf{R}$ (Figure 5.14). The values E and R also correspond to false and true, the possible answers to the question "Is this value needed at runtime?". Section 9.2.1.11 discusses other possible arrangements.

6.1.3 Erasure constraints

Constraint sets are used to capture relationships between erasure annotations in a $\mathsf{TT}_\star$ program. Each constraint has the form $G \to r$, where G is a conjunction of erasure annotations[2] and r is an erasure annotation. Annotations on both sides of a constraint will usually be evars (numbers), but they can also be literal values R or E.

Interpreted in the lattice $\mathsf{E} < \mathsf{R}$, the constraint $G \to r$ stands for $(\bigwedge G) \leqslant r$. Interpreted as a logic program, each constraint is a Horn clause, more specifically a definite clause, and a constraint set is a logic program whose answer set contains exactly the evars that represent the runtime values.

The meaning of the constraint $G \to r$ is that if all annotations in G are used (valued R), then annotation r must be valued R as well, if consistency of erasure annotations is to be preserved. More generally, the value annotated with r must be retained at least as much as most erasable item in G.

Notation I use the following shorthands.

- $r \to s$ stands for $\big\{\{r\} \to s\big\}$ wherever unambiguous.
- $r \leftrightarrow s$ stands for the set of constraints $(r \to s) \cup (s \to r)$.
- $G \wedge r$ stands for $G \cup \{r\}$.
- $G \to R$ stands for $\{G \to r \mid r \in R\}$.
- $R \leftrightarrow S$ stands for $(R \to S) \cup (S \to R)$.

There are several interesting special cases of constraints.

- The constraint $\to r$ means that r must be the top (least erasable) element of the lattice. In terms of logic programming, r does not depend on any assumptions and appears immediately in the answer set, and terms annotated with it are thus necessary for runtime.

  (Compare $\vdash r$ in logic.)

[2] The letter G stands for *guards*, like in Chapter 4.

- The constraint $\rightarrow \mathsf{E}$ represents a contradiction: it says that an erasable value must be available at runtime. A set of constraints is inconsistent iff E appears in the answer set.

  (Compare $\vdash \bot$ in logic.)

- $(G \wedge \mathsf{E}) \rightarrow r$ and $G \rightarrow \mathsf{R}$ and $G \wedge r \rightarrow r$ are tautologies and can be removed from a set of constraints without affecting its answer set.

  (Compare $G, \bot \vdash r$ and $G \vdash \top$ and $G, r \vdash r$ in logic.)

6.2 Reduction rules

Reduction rules of $\mathsf{TT}^{\mathsf{RE}}_{\star}$ do not depend on erasure annotations, and thus we can reuse them for $\mathsf{TT}^{\mathsf{evar}}_{\star}$. Reduction in $\mathsf{TT}^{\mathsf{evar}}_{\star}$ is therefore defined using the same rules as reduction in $\mathsf{TT}^{\mathsf{RE}}_{\star}$ (Section 5.4).

6.3 Type and erasure inference rules

6.3.1 Terms

In Chapter 5, typing judgements are always relative to an erasure annotation: the general form of a typing rule is $\Gamma \vdash T :_r \tau$, where r is the annotation.

For inference rules in this chapter, the general form of typing judgements is $\Gamma \vdash T :_G \tau \mid \Delta$, where G is a (possibly empty) set of zero or more erasure annotations and Δ is a set of erasure constraints relating the erasure annotations that appear in the judgement. The set G represents the *conjunction*, or greatest lower bound, of all annotations (guards) contained within.

All typing rules are designed to take Γ, T, and G as the input and compute τ and Δ as the output (if a solution exists).

6.3.1.1 Typing rules

Figure 6.1 shows the constraint-generating typing rules for terms. They correspond to the checking rules in Chapter 5, Figure 5.15.

Axiom Like with the checking rules, the inference rule Axiom uses type-in-type for simplicity and does not forbid types at runtime.

If we wanted to enforce erasure of types, we could formulate the conclusion of Axiom as $\Gamma \vdash \mathsf{Type} :_G \mathsf{Type} \mid G \rightarrow \mathsf{E}$.

Ref A reference to a bound name marks its definition as retained if it appears in a retained context.

Lam This is the usual lambda rule, except that it also produces the union of constraints coming from its premises.

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

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

$$\frac{\Gamma \vdash \textsc{Def}(n :_s \sigma) \mid \Delta \qquad \Gamma, n :_s \sigma \vdash \rho :_{\{\mathsf{E}\}} \mathsf{Type} \mid \Sigma}{\Gamma \vdash \big((n :_s \sigma) \to \rho\big) :_G \mathsf{Type} \mid \Delta \cup \Sigma}\ \textsc{Pi}$$

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

$$\frac{\Gamma \vdash \textsc{Def}(d) \mid \Delta \qquad \Gamma, d \vdash T :_G \tau \mid \Sigma}{\Gamma \vdash \big(\textbf{let}\ d\ \textbf{in}\ T\big) :_G \tau \mid \Delta \cup \Sigma}\ \textsc{Let} \qquad \frac{\Gamma \vdash T :_G \tau \mid \Delta \qquad \Gamma \vdash \tau \approx \sigma \mid \Sigma}{\Gamma \vdash T :_G \sigma \mid \Delta \cup \Sigma}\ \textsc{Conv}$$

Figure 6.1: Term inference rules for $\mathsf{TT}^{\mathsf{evar}}_{\star}$

Pi Similarly to Lam, this rule first checks the binding and then checks the RHS in the augmented environment, passing on the union of both constraint sets.

App Type-wise, this is the standard dependent application rule. In terms of erasure, the operand X is checked with guards $G \wedge s$, where s is the erasure annotation of the application.

This rule implicitly relies on Conv for conversion checking, which also generates constraints. Typically, these constraints end up included in Σ.

Let Like Lam or Pi, Let checks the definition and then checks the body in the augmented environment.

Conv Since the conversion check may generate constraints (by requiring equality of erasure annotations in corresponding places), the conversion rule involves constraints from both type checking and conversion checking.

6.3.2 Definitions

These rules (Figure 6.2) check well-formedness of definitions, while producing constraints.

DefsBase An empty telescope of definitions is well-formed and produces no constraints.

DefsStep A non-empty telescope is checked by checking its first definition, adding it into the environment, and checking the rest of the telescope.

DefAbstract In abstract definitions, variables and constructors, we check that the proposed type has type Type.

$$\frac{}{\Gamma \vdash \textsc{Defs}(\emptyset) \mid \emptyset}\ \textsc{DefsBase} \qquad \frac{\Gamma \vdash \textsc{Def}(d_1) \mid \Delta \qquad \Gamma, d_1 \vdash \textsc{Defs}\left(\overline{d}^{\,2..n}\right) \mid \Sigma}{\Gamma \vdash \textsc{Defs}\left(\overline{d}^{\,1..n}\right) \mid \Delta \cup \Sigma}\ \textsc{DefsStep}$$

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

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

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

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

FIGURE 6.2: Definition inference rules for $\mathsf{TT}^{\mathsf{evar}}_{\star}$

DEFTERM Definitions with term bodies are allowed to be recursive. Their bodies are checked with the name of the definition present as an abstract variable in the environment. The body is checked with the retention of the whole definition.

DEFCLAUSES A definition consisting of pattern matching clauses is processed clause-by-clause separately. For each clause, we include the (abstract) definition in the environment.

CLAUSE This rule follows the scheme presented in Section 2.1.8.2, and it requires that the names bound in Π are exactly the free pattern variables on the LHS, the LHS is a linear pattern, and that the LHS is forced-pattern-consistent (Definition 5.13).

6.3.3 Patterns

The type and erasure inference rules for patterns follow the same structure as those for terms. While in Chapter 5, the general form of the typing judgement for patterns was $\Gamma; \Pi \vdash P :^q_r \tau$, for erasure inference in this chapter, the judgements will have the form $\Gamma; \Pi \vdash P :^q_G \tau \mid \Delta$. Like in terms, the erasure annotation in the subscript of the colon is replaced with a set of guards G and the judgement is extended with a set of constraints Δ. Figure 6.3 shows the inference rules for patterns.

PATVAR Pattern variables must be bound with retention exactly equal to the retention of the context they appear in.

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

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

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

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

Figure 6.3: Pattern inference rules for $\mathsf{TT}^{\mathsf{evar}}_\star$

PatCtor If an (unforced) constructor is referenced from a pattern, it is a request to check its tag. For this, we must necessarily have the constructor available at runtime and the surrounding pattern must be retained – unless the whole definition is erased.

PatForcedCtor A forced constructor is handled like an ordinary forced pattern.

PatDefName The name of the definition must occur with the retention of the whole definition. Note that during typechecking of its body, f is bound as an abstract variable to allow recursive references but to disallow scrutiny of its body.

PatApp We check X with guards $G \wedge t$, which expresses that if the argument of the application is erased, variable references in X do not cause retention of their definitions.

PatForced Forced patterns do not contain constructors so the value q does not matter. We simply check T as a term in the appropriate context.

PatConv This rule uses the constraint-generating conversion judgement $\Gamma \vdash t \approx \sigma \mid \Sigma$, which is defined in the following Section 6.3.4.

6.3.4 Conversion

Figure 6.4 shows the constraint-generating conversion rules of $\mathsf{TT}^{\mathsf{evar}}_\star$.

Rule ConvRefl uses the relation $\cdot = \cdot \mid \cdot$ (Figure 6.5), which expresses "constrained equality", rather than exact equality. This rule generates constraints equating evars in corresponding places.

$$\frac{\Gamma \vdash \sigma \rightsquigarrow \tau}{\Gamma \vdash \sigma \approx \tau \mid \emptyset}\ \textsc{ConvRed} \qquad \frac{\tau = \sigma \mid \Delta}{\Gamma \vdash \tau \approx \sigma \mid \Delta}\ \textsc{ConvRefl}$$

$$\frac{\Gamma \vdash \sigma \approx \tau \mid \Delta}{\Gamma \vdash \tau \approx \sigma \mid \Delta}\ \textsc{ConvSym} \qquad \frac{\Gamma \vdash \tau \approx \sigma \mid \Delta \quad \Gamma \vdash \sigma \approx \rho \mid \Sigma}{\Gamma \vdash \tau \approx \rho \mid \Delta \cup \Sigma}\ \textsc{ConvTrans}$$

Figure 6.4: Conversion rules of $\mathsf{TT}^{\mathsf{evar}}_{\star}$

$$\frac{}{T = T \mid \emptyset}\ \textsc{CConvRefl} \qquad \frac{F = F' \mid \Phi \qquad X = X' \mid \Xi}{F \mathrel{\widehat{s}} X = F' \mathrel{\widehat{s'}} X' \mid \Phi \cup \Xi \cup s \leftrightarrow s'}\ \textsc{CConvApp}$$

$$\frac{\sigma = \sigma' \mid \Sigma \qquad T = T' \mid \Xi}{\lambda n :_s \sigma.\ T = \lambda n :_{s'} \sigma'.\ T' \mid \Sigma \cup \Xi \cup s \leftrightarrow s'}\ \textsc{CConvLam}$$

$$\frac{\sigma = \sigma' \mid \Sigma \qquad T = T' \mid \Xi}{(n :_s \sigma) \to T = (n :_{s'} \sigma') \to T' \mid \Sigma \cup \Xi \cup s \leftrightarrow s'}\ \textsc{CConvPi}$$

$$\frac{\sigma = \sigma' \mid \Sigma \qquad b = b' \mid \Psi \qquad T = T' \mid \Xi}{(\textbf{let}\ n :_s \sigma = b\ \textbf{in}\ T) = (\textbf{let}\ n :_{s'} \sigma'\ \textbf{in}\ T') \mid \Sigma \cup \Psi \cup \Xi \cup s \leftrightarrow s'}\ \textsc{CConvLet}$$

$$\frac{\forall i.\ \left(C_i = C_i' \mid \Delta_i\right)}{\overline{C} = \overline{C'} \mid \bigcup_i \Delta_i}\ \textsc{CConvBodyClauses} \qquad \frac{b \in \{\textbf{variable}, \textbf{constructor}\}}{b = b \mid \emptyset}\ \textsc{CConvBodyVar}$$

$$\frac{\sigma = \sigma' \mid \Sigma \qquad (\Pi.\ L = R) = (\Pi'.\ L' = R') \mid \Psi}{\left(\Pi, (n :_s \sigma).\ L = R\right) = \left(\Pi', (n :_{s'} \sigma').\ L' = R'\right) \mid \Sigma \cup \Psi \cup s \leftrightarrow s'}\ \textsc{CConvClauseVar}$$

$$\frac{L = L' \mid \Delta \qquad R = R' \mid \Sigma}{(\emptyset.\ L = R) = (\emptyset.\ L' = R') \mid \Delta \cup \Sigma}\ \textsc{CConvClauseBare}$$

$$\frac{}{n = n \mid \emptyset}\ \textsc{CConvPatRef} \qquad \frac{}{\lceil c \rceil = \lceil c \rceil \mid \emptyset}\ \textsc{CConvPatForcedCtor} \qquad \frac{}{\lceil f \rceil = \lceil f \rceil \mid \emptyset}\ \textsc{CConvPatDefName}$$

$$\frac{T = T' \mid \Delta}{[T] = [T'] \mid \Delta}\ \textsc{CConvPatForced} \qquad \frac{F = F' \mid \Phi \qquad X = X' \mid \Xi}{F \mathrel{\widehat{r}} X = F' \mathrel{\widehat{r'}} X' \mid \Phi \cup \Xi \cup r \leftrightarrow r'}\ \textsc{CConvPatApp}$$

Figure 6.5: Constrained equality

6.4 Constraint solving

As shown in Figure 5.2, a program is a term, usually a big let expression. When given a program P in $\mathsf{TT}^{\mathsf{evar}}_{\star}$, we can therefore check it using the rules given above to derive the following judgement.

$$\vdash P :_{\emptyset} \tau \mid \Delta \quad \text{— for some } \Delta \text{ and } \tau$$

The set Δ constrains the valuations of evars with which the program P is well-typed.

In order to find concrete annotations that we can fill into the program to obtain a program in $\mathsf{TT}^{\mathsf{RE}}_{\star}$, which can undergo erasure checking and erasure, we need to find the minimal set of evars that need to be set to R.

There are several ways we could interpret the set Δ and the problem of finding the optimal valuation of evars.

- It can be regarded as a propositional-logic program because each constraint has the form of a definite clause (a Horn clause with no negation in the body). Each annotation r, be it an evar or a specific annotation R and E, can be interpreted as a (ground) atomic proposition "annotation r must be set to R".

  Then finding the minimal set of annotations that need to be set to R amounts to finding the minimal model of the logic program Δ. Since Δ is negation-free, it has exactly one minimal model [GL88; VEK76].

  A model is erasure-consistent iff it does not contain E.

- Mishra-Linger generally presents Δ as a formula in CNF and solving it as a general SAT problem.

  Afterwards, he observes that Δ consists of Horn clauses and unit propagation alone is sufficient for solving it, which is equivalent to the logic-programming approach.

- We can interpret Δ using the complete lattice $L = \mathsf{E} < \mathsf{R}$. If $\mathcal{E}$ is the set of all evars in the program, then the (complete) lattice

  $$L_{\mathcal{E}} := \{v \mid v \in L^{\mathcal{E} \cup \{\mathsf{E},\mathsf{R}\}}, v(\mathsf{R}) = \mathsf{R}\}$$

  is called a lattice of *valuations of erasure annotations*. The least element of $L_{\mathcal{E}}$, $\bot$, maps any annotation to E, except for R, which is mapped to R. The greatest element of $L_{\mathcal{E}}$, $\top$, maps all annotations to R.

  For any valuation of erasure annotations $v : L_{\mathcal{E}}$, we can construct another valuation $f(v) : L_{\mathcal{E}}$ as follows.

  $$f(v)(r) := v(r) \ \vee \bigvee_{(A \to r) \in \Delta} \bigwedge_{r' \in A} v(r')$$

Since $v(r) \leq f(v)(r)$ for all r, we have $v \leq f(v)$ for any valuation v. Because $L_{\mathcal{E}}$ is finite, there must be $k \in \mathbb{N}$ such that $f^k(\bot) = f^{k+1}(\bot)$, and therefore f has a fixed point. Finding the optimal consistent valuation of evars thus amounts to finding the least fixed point of f.

A valuation v is erasure-consistent iff $v(\mathsf{E}) = \mathsf{E}$.

I choose to interpret Δ as a logic program. Then we need to find the minimal set S of annotations such that the following forward-chaining rule holds.

$$\frac{G \subseteq S \qquad (G \rightarrow r) \in \Delta}{r \in S}\ \textsc{ForwardChain}$$

The answer set S is the minimal model of the logic program Δ. All annotations contained in S need to be set to R; all annotations not contained in S can be set to E.

I discuss solving algorithms in Section 6.5.2.

6.4.1 Consistency

A solution S is (erasure-) consistent iff it does not contain E. Since forward chaining finds the minimal model, if this model is not erasure-consistent, there is no erasure-consistent model at all and the compiler must report an error.

A constraint set is erasure-consistent if it has an erasure-consistent solution.

6.4.2 Annotation

After solving the constraints and checking the consistency of the solution, an annotation pass maps programs in $\mathsf{TT}^{\mathsf{evar}}_{\star}$ to programs in $\mathsf{TT}^{\mathsf{RE}}_{\star}$ by mapping the annotations using the function Φ_S defined as follows.

$$\begin{aligned}
\Phi_S(\mathsf{R}) &= \mathsf{R} \\
\Phi_S(\mathsf{E}) &= \mathsf{E} \\
\Phi_S(i) &= \begin{cases} \mathsf{R} & \text{if } i \in S \\ \mathsf{E} & \text{if } i \notin S \end{cases}
\end{aligned}$$

6.5 Discussion

6.5.1 Complexity of erasure inference

The inference algorithm, as presented above, can take quadratic time in the number of evars in the program. It is unknown whether this bound is tight but I will at least provide a few illustrating observations. Consider an environment with the following definitions.

```
data Unit : Type where
  U : Unit
```

$$\mathsf{f} : \mathsf{Unit} \to \mathsf{Unit} \to \mathsf{Unit}$$
$$\mathsf{f}\ \mathsf{U}\ \mathsf{U} = \mathsf{U}$$

In this environment, we check the term T_n defined as follows.

$$\begin{aligned} T_0 &= \mathsf{U} \\ T_{1+i} &= \mathsf{f}\ \mathsf{U}\ T_i \end{aligned}$$

For any given n, the term T_n has length $O(n)$ and it consists of n right-nested applications of $\mathsf{f}\ \mathsf{U}$.

$$T_n = \underbrace{\mathsf{f}\ \mathsf{U}\ \big(\mathsf{f}\ \mathsf{U}\ (\mathsf{f}\ \mathsf{U} \ldots (\mathsf{f}\ \mathsf{U}\ \mathsf{U}) \ldots)\big)}_{n\times}$$

When converted to $\mathsf{TT}^{\mathsf{evar}}_{\star}$, every application is annotated with an evar.

$$T_n = \mathsf{f}\ \widehat{s_n}\ \mathsf{U}\ \widehat{r_n}\ \big(\mathsf{f}\ \widehat{s_{n-1}}\ \mathsf{U}\ \widehat{r_{n-1}}\ (\mathsf{f}\ \widehat{s_{n-2}}\ \mathsf{U}\ \widehat{r_{n-2}} \ldots (\mathsf{f}\ \widehat{s_1}\ \mathsf{U}\ \widehat{r_1}\ \mathsf{U}) \ldots)\big)$$

Assuming that constructor U is bound with erasability r_{U}, we repeatedly use rule App (and Ref) in Figure 6.1 to build a set of constraints for this term that looks like the following.

$$\begin{aligned} & s_n \to r_{\mathsf{U}} \\ & r_n, s_{n-1} \to r_{\mathsf{U}} \\ & r_n, r_{n-1}, s_{n-2} \to r_{\mathsf{U}} \\ & r_n, r_{n-1}, r_{n-2}, s_{n-3} \to r_{\mathsf{U}} \\ & \quad \vdots \\ & r_n, r_{n-1}, \ldots, r_2, s_1 \to r_{\mathsf{U}} \\ & r_n, r_{n-1}, \ldots, r_2, r_1 \to r_{\mathsf{U}} \end{aligned}$$

Even though the above set of constraints can be built relatively cheaply exploiting persistent data structures and sharing, the logical size of this set of constraints is quadratic and with linear-time algorithms [DG84], solving the set of constraints would also take quadratic time in the number of evars in the program.

Various constraint solving algorithms are discussed below in Section 6.5.2.

6.5.2 Efficiency of constraint solving

Definition 6.1 (Reduction of constraints)**.** For a set of constraints Δ and a set of evars N, we define Δ^N, reduction of Δ by N, as follows.

$$\Delta^N = \{G \setminus N \to r \mid (G \to r) \in \Delta, G \neq \emptyset\}$$

Quadratic complexity of naïve unit propagation The straightforward algorithm of iterated unit propagation has (at least) a quadratic time complexity because in each

iteration, it traverses all constraints.

$$\textsc{UnitProp}(\Delta) = \begin{cases} \emptyset & \text{if } N = \emptyset \\ N \cup \textsc{UnitProp}(\Delta^N) & \text{otherwise} \end{cases}$$

where

$$N = \{r \mid (\rightarrow r) \in \Delta\}$$

The worst-case scenario is represented by the following set of constraints.

$$\Delta_{\text{worst}} = \{\rightarrow 1\} \cup \bigcup_i \{i \rightarrow i+1\}$$

The set Δ_{worst} represents a long linear chain of implications and even if we use a data structure that allows efficient lookup of constraints of the form $(\rightarrow r)$, repeated *reduction* will take $O(n^2)$ time in total.

Index of constraints In my implementation, I use an *index* that maps every evar to the set of constraints where the evar appears among the preconditions. More precisely, I give each constraint a unique number, and then the index maps evars to sets of numbers; $I : \mathcal{E} \rightarrow 2^{\mathbb{N}}$.

Assuming that $\Delta = \Sigma \cup \Xi$, where Ξ does not contain constraints with indices in $I[N]$, we can define reduction by N using index I as follows.

$$\Delta_I^N = \Sigma^N \cup \Xi$$

This means that we reduce only the relevant part of Δ. This is more efficient and we can use Δ_I^N instead of Δ^N in the iterated unit propagation algorithm.

As clauses are removed from Δ by reduction, the index I will contain more and more constraint numbers that are not present in Δ anymore. In my implementation, I ignore this fact and keep the index I constant throughout all iterations.

My implementation pre-reduces constraint sets for each definition in a program separately to provide a more compact representation, and therefore the constraint solver is invoked many times, for constraint sets of various sizes, including the constraint set for the definition of `main`. In my benchmark programs, there is (surprisingly) almost no speedup gained by using the non-indexed naïve solver for small constraint sets, and the solver gets much slower if threshold of what is a "small" constraint set is defined incorrectly. My implementation therefore always uses the indexing solver, even for small problems.

Last-precondition algorithm Mishra-Linger [ML08] goes farther and implements a solver based on [Mos+01]. The algorithm is based on the observation that we need not visit a constraint *every* time that one of its preconditions is satisfied. It is sufficient to visit a constraint once *the last* precondition is satisfied.

The algorithm is called "two watched literals" because in the CNF representation of the constraints, two literals are distinct – the last precondition (that is being unit-propagated), and the other literal that will turn out to be the consequence.

However, I prefer to think of the algorithm in terms of Horn clauses, where we single out only *one* precondition to watch because the other literal – the consequence – is special by the nature of Horn clauses.

In this algorithm, the index of constraints above is modified so that it contains a mapping from evars to the clauses *where that evar is watched*, which is generally a smaller set than the set of clauses where that evar occurs among the preconditions at all.

My implementation used to select evars to watch based on the number of clauses where the evar appears as the consequence, using a pre-computed mapping from evars to numbers. This seems to be slower than arbitrarily selecting the first precondition from the set as the watched precondition.

The unit propagation step works as above, except that when we visit a constraint and reduce it according to Definition 6.1, there are two possibilities.

1. Constraint $G \to r$ reduces to $G \setminus N \to r$ where $G \setminus N = \emptyset$. In this case, this constraint will be picked up in the next iteration of unit propagation and we need not do anything here.

2. Constraint $G \to r$ reduces to $G \setminus N \to r$ where $G \setminus N \neq \emptyset$. In this case, the watched evar is not the last precondition of the constraint and we need to select a new precondition to watch.

This approach traverses even fewer constraints than the original index-based method, but at the expense of more bookkeeping.

In my implementation, any attempt to improve the plain indexed algorithm, including this last-precondition algorithm, leads to a slower solver in the benchmarks and I expect that the benefits of the theoretically better algorithms will show only in large programs.

Exploiting equivalences In the inference rules, many constraints are actually equivalences between evars. If we merge evars that should be equivalent, instead of merely producing constraints between them, it could reduce the size of the constraint set, although it would not asymptotically improve the quadratic program above.

Other approaches Dowling and Gallier present linear-time algorithms for testing the satisfiability of propositional Horn formulae [DG84]. One of their algorithms ("Linear-time refining of Algorithm 1") is similar to the above approach with an index. They present also other approaches using graph algorithms.

6.6 Correctness

Here I prove that erasure inference is sound, complete, and optimal with respect to the typing rules of $\mathsf{TT}^{\mathsf{RE}}_{\star}$ given in Chapter 5.

Let $\Phi : \mathbb{N} \cup \{\mathsf{R}, \mathsf{E}\} \to \{\mathsf{R}, \mathsf{E}\}$ be a valuation of evars 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 and as $\Phi : \mathsf{TT}^{\mathsf{evar}}_{\star} \to \mathsf{TT}^{\mathsf{RE}}_{\star}$ to mean substitution for evars in a $\mathsf{TT}^{\mathsf{evar}}_{\star}$ term, yielding a fully erasure-annotated $\mathsf{TT}^{\mathsf{RE}}_{\star}$ term.

Definition 6.2 (Semantics of erasure constraints)**.** Evar valuation Φ models a constraint set Δ if for each constraint $(G \to r) \in \Delta$, retention of G implies retention of r.

$$\frac{\forall (G \to r) \in \Delta.\ \big(\Phi(G) \leqslant \Phi(r)\big)}{\Phi \models \Delta}$$

6.6.1 Soundness

Lemma 6.1.

$$\frac{\Phi \models G \to r}{\Phi(G) \leqslant \Phi(r)}$$

Proof. Directly from Definition 6.2. □

Lemma 6.2.

$$\frac{\Phi \models r \leftrightarrow s}{\Phi(r) = \Phi(s)}$$

Proof. From Definition 6.2 and the definition of $\leftrightarrow$ (Section 6.1.3). □

Lemma 6.3.

$$\frac{\Phi \models \Delta \cup \Sigma}{\Phi \models \Delta}$$

Proof. From Definition 6.2. □

Lemma 6.4.

$$\frac{\sigma = \tau \mid \Delta \qquad \Phi \models \Delta}{\Phi(\sigma) = \Phi(\tau)}$$

Proof. By induction on the proof of $\sigma = \tau \mid \Delta$, using Lemma 6.3 and Lemma 6.2. □

Lemma 6.5.

$$\frac{\Gamma \vdash \tau \approx \sigma \mid \Delta \qquad \Phi \models \Delta}{\Phi(\Gamma) \vdash \Phi(\tau) \approx \Phi(\sigma)}$$

Proof. By induction on the proof of conversion.

ConvRed Reduction does not depend on erasure annotations so $\Phi(\Gamma) \vdash \Phi(\sigma) \rightsquigarrow \Phi(\tau)$ and we can re-apply ConvRed for $\mathsf{TT}^{\mathsf{RE}}_{\star}$.

ConvRefl By Lemma 6.4, $\Phi(\tau) = \Phi(\sigma)$ and we can apply ConvRefl for $\mathsf{TT}^{\mathsf{RE}}_{\star}$.

ConvSym By induction.

ConvTrans By induction and Lemma 6.3.

□

Theorem 6.1 (Soundness of erasure inference)**.** *Any valuation of evars Φ that models the set of constraints computed by erasure inference, as given by the rules in Section 6.3, is erasure-correct, as given by the rules in Section 5.5.*

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

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

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

Proof. By (mutual) induction on the typing derivations.

Terms

Axiom Trivially.

Ref By inversion of Ref for $\mathsf{TT}^{\mathsf{evar}}_{\star}$, we have $(n :_s \tau = b) \in \Gamma$ such that $\Phi \models G \longrightarrow s$. By Lemma 6.1, $\Phi(G) \leqslant \Phi(s)$. Since $\big(n :_{\Phi(s)} \Phi(\tau) = \Phi(b)\big) \in \Phi(\Gamma)$, we can use Ref for $\mathsf{TT}^{\mathsf{RE}}_{\star}$ to obtain the desired typing judgement.

Lam By induction and Lemma 6.3.

Pi By induction and Lemma 6.3.

App By inversion of App for $\mathsf{TT}^{\mathsf{evar}}_{\star}$ and induction, we have $\Phi(\Gamma) \vdash \Phi(F) :_{\Phi(G)} \big(n :_{\Phi(t)} \Phi(\sigma)\big) \longrightarrow \Phi(\rho)$ and also $\Phi(\Gamma) \vdash \Phi(X) :_{\Phi(G) \wedge \Phi(s)} \Phi(\sigma)$ such that $\Phi \models t \leftrightarrow s$. By Lemma 6.2, we have $\Phi(t) = \Phi(s)$, and thus we can use App for $\mathsf{TT}^{\mathsf{RE}}_{\star}$ to obtain the desired conclusion.

Let By induction and Lemma 6.3.

Conv By induction, Lemma 6.3, and Lemma 6.5.

Patterns

PatVar By Lemma 6.2, we have $\Phi(s) = \Phi(G)$, and by inversion of PatVar for $\mathsf{TT}^{\mathsf{evar}}_{\star}$, we have $\left(n :_{\Phi(s)} \Phi(\sigma)\right) \in \Phi(\Pi)$. Apply PatVar for $\mathsf{TT}^{\mathsf{RE}}_{\star}$.

PatCtor By Lemma 6.1, we have $\Phi(q) \leqslant \Phi(G) \wedge \Phi(s)$, and thus $\Phi(q) \leqslant \Phi(G)$ and $\Phi(q) \leqslant \Phi(s)$. By Inversion of PatCtor for $\mathsf{TT}^{\mathsf{evar}}_{\star}$, we have $\left(n :_{\Phi(s)} \Phi(\sigma) = \textbf{constructor}\right) \in \Phi(\Gamma)$. Apply PatCtor for $\mathsf{TT}^{\mathsf{RE}}_{\star}$.

PatApp Like App for terms above.

PatForced By induction.

PatConv By induction and Lemma 6.5.

Definitions By induction on the proof of well-typedness of the definition, using Lemma 6.3. This case includes rule Clause.

Telescopes We proceed by induction on the length of the telescope, using Lemma 6.3.

□

6.6.2 Completeness

Lemma 6.6.

$$\frac{\Phi \models \Delta \qquad \Phi \models \Sigma}{\Phi \models \Delta \cup \Sigma}$$

Proof. From the definition of $\models$, thanks to the fact that both Δ and Σ are modelled by the same valuation Φ. □

Lemma 6.7.

$$\frac{\Phi(r) = \Phi(s)}{\Phi \models r \leftrightarrow s}$$

Proof. From Definition 6.2 and the definition of $\leftrightarrow$ (Section 6.1.3). □

Lemma 6.8.

$$\frac{\Phi(T) = \Phi(T')}{\exists \Delta. \qquad T = T' \mid \Delta \qquad \Phi \models \Delta}$$

Proof. By induction on the structure of $\Phi(T)$ (and therefore $\Phi(T')$, too). For every syntactic element, $\cdot = \cdot \mid \Delta$ is defined by exactly one rule, and we can verify that Δ contains only constraints of the form $r \leftrightarrow s$ such that $\Phi(r) = \Phi(s)$, and therefore $\Phi \models r \leftrightarrow s$, and thus $\Phi \models \Delta$. □

Lemma 6.9.

$$\frac{\Phi(\Gamma) \vdash \Phi(\sigma) \approx \Phi(\tau)}{\exists \Delta. \quad \Gamma \vdash \sigma \approx \tau \mid \Delta \qquad \Phi \models \Delta}$$

Proof. By induction on the derivation of conversion.

ConvRefl From Lemma 6.8.

ConvRed Reduction does not depend on erasure annotations so $\Gamma \vdash \sigma \rightsquigarrow \tau$ and we can re-apply ConvRed for $\mathsf{TT}^{\mathsf{evar}}_{\star}$.

ConvSym By induction.

ConvTrans By induction and Lemma 6.6.

□

Theorem 6.2 (Completeness of erasure inference)**.** *Any erasure-correct valuation of evars typechecks as a $TT^{evar}_{\star}$ program and is a solution to the constraints inferred from the program.*

Like Theorem 6.1, also this theorem consists of four mutually recursive components.

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

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

Proof. By (mutual) induction on the typing derivations.

Terms

Axiom Trivially since $\Phi \models \emptyset$ for any Φ.

Ref By inversion of Ref for $\mathsf{TT}^{\mathsf{RE}}_{\star}$, there is evar s such that $(n :_s \tau = b) \in \Gamma$ and $\Phi(G) \leqslant \Phi(s)$. The conclusion holds because we can apply Ref for $\mathsf{TT}^{\mathsf{evar}}_{\star}$ and we also have $\Phi \models G \longrightarrow s$ by definition of $\models$.

Lam By inversion of Lam for $\mathsf{TT}^{\mathsf{RE}}_{\star}$, $\Phi(\Gamma) \vdash \Phi(\sigma) :_{\Phi(\mathsf{E})} \Phi(\mathsf{Type})$ and $\Phi(\Gamma, n :_s \sigma) \vdash \Phi(T) :_{\Phi(G)} \Phi(\rho)$ such that $\Phi(\tau) = \Phi\big((n :_s \sigma) \longrightarrow \rho\big)$.

By induction, we obtain $\Gamma \vdash \sigma :_{\mathsf{E}} \mathsf{Type} \mid \Delta$ such that $\Phi \models \Delta$ and $\Gamma, n :_s \sigma \vdash T :_G \rho \mid \Sigma$ such that $\Phi \models \Sigma$. By Lemma 6.6, $\Phi \models \Delta \cup \Sigma$, and we also reapply Lam for $\mathsf{TT}^{\mathsf{evar}}_{\star}$.

Pi Like Lam.

App where $T = F \,\widehat{s}\, X$. By inversion of App for $\mathsf{TT}^{\mathsf{RE}}_{\star}$ and induction, we have $\Gamma \vdash F :_G (n :_s \sigma) \longrightarrow \rho \mid \Delta$ and $\Gamma \vdash X :_{G \wedge s} \sigma \mid \Sigma$ such that $\Phi \models \Delta$ and $\Phi \models \Sigma$.

By Lemma 6.6, $\Phi \models \Delta \cup \Sigma$. We can also apply App for $\mathsf{TT}^{\mathsf{evar}}_{\star}$ with $s = t$, and thus $\Phi \models s \leftrightarrow t$ by Lemma 6.7.

We therefore have the requested typing judgement, and by Lemma 6.6, Φ models the resulting constraint set.

Let By induction and Lemma 6.6.

Conv By inversion of Conv, induction, and Lemma 6.9, we obtain $\Gamma \vdash T :_G \sigma \mid \Delta$ such that $\Phi \models \Delta$ and $\Gamma \vdash \sigma \approx \tau \mid \Sigma$ such that $\Phi \models \Sigma$. We can therefore apply Conv and by Lemma 6.6, $\Phi \models \Delta \cup \Sigma$.

Patterns

PatVar By inversion of PatVar for $\mathsf{TT}^{\mathsf{RE}}_{\star}$, we have $\Phi(G) = \Phi(s)$. We can reapply PatVar for $\mathsf{TT}^{\mathsf{evar}}_{\star}$ and by Lemma 6.7, $\Phi \models \{s\} \leftrightarrow G$.

PatCtor By inversion of PatCtor for $\mathsf{TT}^{\mathsf{RE}}_{\star}$, we have $\Phi(q) \leqslant \Phi(G)$ and $\Phi(q) \leqslant \Phi(s)$. We have therefore $\Phi(q) \leqslant \Phi(G \wedge s)$, and $\Phi \models \{q\} \leftrightarrow (G \wedge s)$.

PatApp Like App for terms.

PatForced By induction.

PatConv Like Conv for terms, by Lemma 6.9.

Definitions By induction on the proof of well-typedness of the definition, using Lemma 6.6. This includes Clause.

Telescopes By induction on the size of the telescope.

□

6.6.3 Optimality

Definition 6.3 (Model size)**.** The size of Φ is the number of evars mapped to R.

$$|\Phi| := |\{r \mid r \neq \mathsf{R} \wedge \Phi(r) = \mathsf{R}\}|$$

Theorem 6.3 (Optimality of erasure inference)**.** *Let $\Gamma \vdash T :_G \tau \mid \Delta$, let Φ be computed from Δ using forward chaining as described in Section 6.4, and let Φ' be any consistent valuation, which means that $\Phi'(\Gamma) \vdash \Phi'(T) :_{\Phi'(G)} \Phi'(\tau)$. Then $|\Phi| \leq |\Phi'|$.*

In other words, the erasure algorithm given in this chapter finds a solution with the minimal number of annotations set to R, compared to any other solution that would typecheck using the rules in Section 5.5.

Proof. By completeness (Theorem 6.6.2), any valid solution, including Φ', models Δ. For each answer set S, the corresponding valuation Φ_S has the same size $|\Phi_S| = |S|$. Since the answer set in Section 6.4 is defined as the minimal model of Δ, $|\Phi| \leq |\Phi'|$. □

Remark 6.1. The solution inferred by the algorithm given in this chapter is "optimal" in the same sense as Mishra-Linger's algorithm [ML08] – optimal within an approximation of the problem of erasure: the approximation defined by the typing rules of $\mathsf{TT}^{\mathsf{RE}}_{\star}$, and optimal in terms of the *number* of binders erased.

However, it is important to distinguish between the notion of optimality as given by Mishra-Linger and the "true" optimality of erasure inference, which is of course an undecidable problem.

For examples of programs where this erasure inference algorithm produces sub-optimal results in the latter sense, see Chapter 7.

Chapter 7

Extensions

This chapter lists some possible improvements to the core language that was introduced in Chapter 5. These improvements are motivated by particular issues in interaction between erasure and programming.

I discuss that some functions become identities after erasure, case trees in the core calculus, erasure polymorphism, ways to make erasure inference non-whole-program, that erasure interacts well with I/O and FFI, and sketch how erasure could co-exist with irrelevance.

7.1 Identity optimisation

After erasure, many functions become identity functions; the typical example being $\mathsf{subst} : (f : a \to \mathsf{Type}) \to x \equiv y \to f\ x \to f\ y$ (Listing 2.21 in Section 2.2.3.1),

$$\begin{aligned} &\mathsf{subst}\ f\ \mathsf{Refl}\ x = x && \textit{— unerased}\\ &\mathsf{subst}\ x = x && \textit{— erased} \end{aligned}$$

or $\mathsf{embed} : \mathsf{Fin}\ n \to \mathsf{Fin}\ (\mathsf{S}\ n)$ (Section 2.2.1.2).

$$\begin{aligned} &\mathsf{embed}\ \ \mathsf{FZ}\ \ \ = \mathsf{FZ} && \textit{— erased clause 1}\\ &\mathsf{embed}\ \ (\mathsf{FS}\ x) = \mathsf{FS}\ (\mathsf{embed}\ x) && \textit{— erased clause 2} \end{aligned}$$

This has been recognised as a problem also by McBride [McB14c].

My implementation uses a simple inductive checker whether an erased recursive function is an obvious identity and then for each I recognised as an identity function, it replaces all occurrences of $(I\ X)$ with X. This happens after erasure, in the untyped $\mathsf{TT}^{\square}_{\star}$ stage.

7.2 Case trees

Section 2.1.3.2 mentions that case trees are a good intermediary between high-level pattern matching and low-level sequential code since there are established methods of compiling pattern matching to case trees [Aug85; Wad87a].

7.2.1 Case trees in a core calculus

However, case trees would also be a good *replacement* for pattern clauses as the pattern matching facility of a core language, where pattern clauses in the surface would need to be compiled into case trees in the core language.

Since case trees are more explicit and more operational, they might be more suitable for a core calculus. The safety net of type checking would thus cover a larger part of the compiler – at least the clauses-to-trees transformation, which a compiler would likely perform at some stage anyway. Coverage checking, checking forced patterns and checking overlapping patterns (Section 2.1.5.1) would become more straightforward, as one could use established methods [GMM06] directly. Finally, Section 2.1.5.1 illustrates that some inconsistent programs are inexpressible in the form of case trees.

Originally, $\mathsf{TT}_\star$ had case trees at its core. However, since these were type checked by translation to pattern clauses – and this dissertation is focused on type checking – I chose ease and simplicity of type checking over other considerations.

7.2.2 Case trees in dependently typed languages

Different languages introduce different variants of case trees which differ in approach and flexibility: some of them allow nested patterns, some don't; some of them perform dependent matching with rewriting in types, some produce explicit equalities for type family indices.

7.2.2.1 Idris

The core calculus of Idris, TT, is based on pattern matching clauses. However, the compiler internally translates pattern matching to baseline case trees (Section 2.1.3.2) with the following syntax, where D is a definition, T is a case tree, R is an expression/term ("the RHS"), B is a case branch, c is a constructor name, f is a function name, x is a variable name.

$$
\begin{array}{lcl}
D & ::= & f = \lambda \overline{x}.\ T \\
T & ::= & R \\
 & \mid & \textbf{case}\ x\ \textbf{of}\ \overline{B} \\
B & ::= & c\ \overline{x} \Rightarrow T
\end{array}
$$

These case trees have two important restrictions.

- Only (pattern) variables can be inspected, case-splitting on complex terms is disallowed.
- A case tree is not a term: every pattern matching function definition has the form $f = \lambda \overline{x}.\ T$, where T is a case tree referring to the variables bound in the lambda, and containing terms in the leaves.

Case trees in the intermediate representation of Idris are restricted in this way[1] because this is the form of the output of the pattern matching compiler [Wad87a]. *Split trees*, as described by Goguen, McBride, McKinna [GMM06], also follow this structure. Finally, it happens to be a convenient form for other purposes, such as translating case trees back to pattern clauses (Section 7.2.3).

Therefore it seems that this form of case trees might be the minimal form required to implement pattern matching.

7.2.2.2 Coq

Coq features **match** expressions, which can be nested to form case trees. Coq's match expressions do not have any of the syntactic restrictions of the case trees in the IR of Idris and are briefly characterised in Section 2.1.8.2.

7.2.2.3 Zombie

Data types in Zombie do not have indices, only parameters [Sjö15]. Indices are implemented using Henry Ford indexing[2] [McB00] – as *parameters* of the type constructor, together with extra equality arguments in the data constructors, as shown in the following example.

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Vec}\ (n : \mathbb{N})\ (a : \mathsf{Type})\ \textbf{where} \\
\quad \mathsf{Nil}\ \ \ \textbf{of}\ \ (eq_{n=0} : n \equiv 0) \\
\quad \mathsf{Cons}\ \ \textbf{of}\ \ (m : \mathbb{N})\ (eq_{n=\mathsf{Succ}\ m} : n \equiv \mathsf{Succ}\ m)\ (x : a)\ (xs : \mathsf{Vec}\ m\ a)
\end{array}
$$

The data elimination facility of Zombie is case expressions, where each match introduces new equalities into the environment.

$$
\begin{array}{l}
\mathsf{vlen} : (a : \mathsf{Type}) \to (n : \mathbb{N}) \to \mathsf{Vect}\ n\ a \to \mathbb{N} \\
\mathsf{vlen}\ a\ n\ xs = \textbf{case}\ xs\ [eq_{xs}]\ \textbf{of} \\
\qquad\qquad\qquad \begin{array}{lcl}
\mathsf{Nil}\ eq_{n=0} & \Rightarrow & rhs_1 \\
\mathsf{Cons}\ m\ eq_{n=\mathsf{Succ}\ m}\ x\ xs & \Rightarrow & rhs_2
\end{array}
\end{array}
$$

The environment of rhs_1 includes the following equalities.

$$
\begin{array}{lcl}
eq_{n=0} & : & n \equiv 0 \\
eq_{xs} & : & xs \equiv \mathsf{Nil}\ eq_{n=0}
\end{array}
$$

The environment of rhs_2 includes the following equalities.

$$
\begin{array}{lcl}
eq_{n=\mathsf{Succ}\ m} & : & n \equiv \mathsf{Succ}\ m \\
eq_{xs} & : & xs \equiv \mathsf{Cons}\ m\ eq_{n=\mathsf{Succ}\ m}\ x\ xs
\end{array}
$$

[1] Case trees in the intermediate representation are distinct from *case expressions* in the surface language. Case expressions do not have any of these syntactic restrictions.

[2] "Any customer can have a car painted any color that he wants so long as it is black."

The equality eq_{xs} is the "smart case" equality and it witnesses the different form of the scrutinee in each case branch. The remaining equalities are the Henry-Ford index equalities contained within the constructors.

For checkability, the core language contains explicit type casts using the equalities from the environment and elaborated from the surface language automatically using congruence closure. This is possible because conversion in Zombie does not include β reduction by default [Sjö15].

7.2.2.4 Cayenne

Data types in Cayenne are defined in a slightly unusual way but case expressions are ordinary enough – they are essentially baseline case trees, in the terminology introduced in Section 2.1.3.2, and similar case trees are used in the intermediate representation(s) of Idris (Section 7.2.2.1). The following is an example of a case expression, as given by Augustsson.

```
case x of
  (True)  → 1;
  (False) → "Hello";
  :: (case x of
        (True)  → Int;
        (False) → String;
   )
```

The case expression requires an explicit type annotation because it is not possible to infer it in the general case [Aug99]. This type annotation is equivalent to the annotations necessary for top-level functions in Idris and match expressions in Coq (Section 2.1.8.2).

7.2.2.5 Epigram

Pattern matching definitions in Epigram [MM04; McB05; GMM06] contain both case trees and pattern clauses *at the same time*. Here's an example definition.

$$
\begin{array}{lllll}
\textbf{let}\ \dfrac{x : \mathbb{N}}{\mathsf{fib}\ x : \mathbb{N}}\ \mathsf{fib} & x & \Leftarrow & & \mathbb{N}\text{-rec}\ x\ \{ \\
\quad \mathsf{fib} & x & \Leftarrow & & \mathbb{N}\text{-case}\ x\ \{ \\
\quad\quad \mathsf{fib} & \mathsf{Z} & \mapsto & \mathsf{Z} & \\
\quad\quad \mathsf{fib} & (\mathsf{S}\ y) & \Leftarrow & & \mathbb{N}\text{-case}\ y\ \{ \\
\quad\quad\quad \mathsf{fib} & (\mathsf{S}\ \mathsf{Z}) & \mapsto & \mathsf{Z} & \\
\quad\quad\quad \mathsf{fib} & \big(\mathsf{S}\ (\mathsf{S}\ z)\big) & \mapsto & \mathsf{fib}\ z + \mathsf{fib}\ (\mathsf{S}\ z) & \}\}\}
\end{array}
$$

Syntactically, the declaration is organised in a tree shape with eliminators in the internal nodes ($\mathbb{N}$-rec, $\mathbb{N}$-case), given by the programmer, and pattern clauses in the leaves, with LHSs derived by Epigram.

This is a very interesting design choice. While $\mathsf{TT}_\star$ needs explicit marking of forced patterns and forced constructors in pattern clauses and translation from pattern clauses to case trees before code generation, Epigram can obtain all this information from the structure of the case trees. Left-hand sides of pattern clauses can then be provided only as a friendly reminder for the human programmer where they are in the case tree but they do not contain any new information.

In Section 2.1.4.3, I argue that the programmer must be responsible for the choice of forced patterns. Epigram presents a way of making that choice naturally by choosing the structure of the case tree.

Epigram allows other eliminators than the usual induction principles split into X-elim and X-rec but we will focus on the fact that we can derive pattern clauses from case trees.

7.2.3 Converting case trees to pattern matching clauses

This section describes a way to convert baseline case trees (as described in Section 2.1.3.2) into pattern matching clauses. This is useful for type checking of definitions given by case trees, without the need of unification in the type checker.

The translation is based essentially on the same principles as introduced by Goguen, McBride, McKinna [GMM06] in their notion of *split trees*, as used also by Cockx [Coc17]. We start with the most general clause in the root of the tree and specialise it on the way to the leaves. Each leaf then corresponds to a pattern clause in the output sequence.

7.2.3.1 Syntax

In Section 7.2.2.1, I gave the syntax for baseline case trees. Here, D stands for definitions, T is a case tree, R is a term (the RHS), f is the name of the function being defined, c is a constructor name, x is a variable name. Here, let us add type annotations to all binders and also extend the syntax of branches with a forced-pattern branch.

$$
\begin{array}{rcl}
D & ::= & f = \lambda\overline{(x : R)}.\ T \\
T & ::= & R \\
 & \mid & \textbf{case}\ x\ \textbf{of}\ \overline{B} \\
B & ::= & c\ \overline{(x : R)} \Rightarrow T \\
 & \mid & [R] \Rightarrow T
\end{array}
$$

We want to translate any definition given in the syntax above to the syntax of pattern clauses, where a definition is a sequence of clauses, a clause contains explicit type annotations for pattern variables with a pattern on the left-hand side and a term on the right-hand side, and a pattern is either a name (of a constructor or a variable), a forced pattern, or pattern application. This is exactly the syntax of pattern clauses

in $\mathsf{TT}_\star$ (5.2).

$$\begin{aligned} D &::= C^\star \\ C &::= \overline{(x : R)}.\ P = R \\ P &::= x \mid [R] \mid P\ P \end{aligned}$$

7.2.3.2 Translation to pattern clauses

Definition 7.1 (Substitution of patterns into patterns)**.** Substitution of *patterns* uses the symbol $\mapsto_\mathsf{P}$, unlike standard substitution of *terms*, which we write $\mapsto$.

$$\begin{aligned} n[x \mapsto_\mathsf{P} P] &:= n && \text{— if } n \neq x \\ x[x \mapsto_\mathsf{P} P] &:= P \\ [R][x \mapsto_\mathsf{P} P] &:= \left[R[x \mapsto |P|]\right] \\ (F\ X)[x \mapsto_\mathsf{P} P] &:= F[x \mapsto_\mathsf{P} P]\ X[x \mapsto_\mathsf{P} P] \\ \lfloor f \rfloor[x \mapsto_\mathsf{P} P] &:= \lfloor f \rfloor && \text{— } f \neq x \\ \lceil c \rceil[x \mapsto_\mathsf{P} P] &:= \lceil c \rceil \end{aligned}$$

Pattern-to-term conversion, $|\cdot|$, is defined in Definition 5.9.

Definition 7.2 (Substitution into typing environments)**.** Substitution in typing environments $\Pi[x \mapsto_\mathsf{E} \Delta/X]$ is defined as follows. It replaces the binding of x with a sequence of binders Δ, and all references to x in the terms to the right are replaced with X.

$$\left(\overline{(y : R_y)}, (x : R), \overline{(z : R_z)}\right)[x \mapsto_\mathsf{E} \Delta/X] := \overline{(y : R_y)}, \Delta, \overline{(z : R_z[x \mapsto X])}$$

Translating function definitions The translation procedure is purely syntactical and I will use the notation $[\![\cdot]\!]$ to stand for it. I use $[\![\cdot]\!]_\mathsf{D}$ for whole definitions, $[\![\cdot]\!]_\mathsf{T}$ for case trees, and $[\![\cdot]\!]_\mathsf{B}$ for branches.

A whole definition D given by case trees is translated to pattern clauses as follows.

$$\left[\!\!\left[f = \lambda\overline{(x : R)}.\ T \right]\!\!\right]_\mathsf{D} := \left[\!\!\left[\overline{(x : R)}.\ \lfloor f \rfloor\ \overline{x} = T \right]\!\!\right]_\mathsf{T}$$

The intermediate step $[\![\Pi.\ P = T]\!]_\mathsf{T}$ has three inputs, an environment of pattern variables Π, a pattern P, and a case tree T, and it returns a sequence of pattern clauses.

Translation of case tree leaves, where R is a term, is straightforward – we form a pattern clause using the given inputs.

$$[\![\Pi.\ P = R]\!]_\mathsf{T} := \Pi.\ P = R$$

Translation of case splits concatenates translations of the individual branches, which themselves return sequences of clauses. (Splits with a single branch are translated

using a rule given further below.)

$$\left[\!\left[\Pi.\ P = \left(\textbf{case}\ x\ \textbf{of}\ \overline{B_i}^{\,i\in 1..n}\right)\right]\!\right]_{\mathsf{T}} := \textsc{Concat}\left(\overline{\left[\!\left[\Pi.\ P = (x \mid B_i)\right]\!\right]_{\mathsf{B}}}^{\,i\in 1..n}\right) \quad \text{if } n > 1$$

Translation of ordinary case branches is given by the following rule.

$$\begin{aligned}&\left[\!\left[\Pi.\ P = \left(x \mid c\ \overline{(y : R)} \Rightarrow T\right)\right]\!\right]_{\mathsf{B}} \\ &\quad := \left[\!\left[\Pi[x \mapsto_{\mathsf{E}} \overline{(y : R)}/c\ \overline{y}].\ P[x \mapsto_{\mathsf{P}} c\ \overline{y}] = T\right]\!\right]_{\mathsf{T}}\end{aligned} \tag{7.1}$$

Translation of forced case branches is given by he following rule, where R' is the term to which the pattern is forced. The binding of x is replaced with the empty sequence $\emptyset$, which means it is deleted and all occurrences of x to the right of the binding are replaced with R'.

$$\left[\!\left[\Pi.\ P = (x \mid [R'] \Rightarrow T)\right]\!\right]_{\mathsf{B}} := \left[\!\left[\Pi[x \mapsto_{\mathsf{E}} \emptyset/R'].\ P[x \mapsto_{\mathsf{P}} [R]] = T\right]\!\right]_{\mathsf{T}} \tag{7.2}$$

Finally, if the pattern is covering, single-branch trees are translated specially using forced constructors. If the pattern is not covering, we must use rule 7.1 here instead.

$$\begin{aligned}&\left[\!\left[\Pi.\ P = \left(\textbf{case}\ x\ \textbf{of}\ c\ \overline{(y : R)} \Rightarrow T\right)\right]\!\right]_{\mathsf{T}} \\ &\quad := \left[\!\left[\Pi[x \mapsto_{\mathsf{E}} \overline{(y : R)}/c\ \overline{y}].\ P[x \mapsto_{\mathsf{P}} \lceil c \rceil\ \overline{y}] = T\right]\!\right]_{\mathsf{T}}\end{aligned} \tag{7.3}$$

Translation of a case branch involves substitution into the pattern variable environment and into the pattern, as defined in Definitions 7.2 and 7.1. This corresponds to validation of internal nodes of split trees, as defined by Goguen, McBride and McKinna [GMM06], and they even use the same notion of substitution into the environment of pattern variables.

7.2.3.3 Choice of scoping of inspected variables

In the translation, as defined above, case inspection of a variable entirely eradicates this variable from the context and any patterns that might refer to it, and does not substitute for it in the case tree on the RHS.

In practice, this means that we cannot refer to the variable from any leaf of the case tree that lies below the case split – if we do, the resulting pattern clause will contain a reference to an unbound variable.

I found this choice the easiest to reason about but there are two other choices.

Direct references We could modify the environment substitution rule in Definition 7.2 to *not* remove the binding for x.

$$\left(\overline{(y : R_y)}, (x : R), \overline{(z : R_z)}\right)[x \mapsto_{\mathsf{E}} \Delta/X] := \overline{(y : R_y)}, (x : R), \Delta, \overline{(z : R_z[x \mapsto X])}$$

This would allow an efficient implementation of as-patterns, as introduced in Haskell [Jon03], by allowing the leaves to refer directly to the individual pattern variables.

In the translation process and in the resulting pattern clauses, this approach would probably require explicit bookkeeping of all variables that are both referred to directly *and* also case-split, most likely as explicit as-patterns.

Substitution on the RHS We could also modify the translation rules given by Equations 7.1, 7.2 and 7.3 to also substitute for x in the tree T on the RHS. Equation 7.1 would be changed as follows.

$$\begin{aligned}&\left[\!\left[\Pi.\ P = \left(x \mid c\ \overline{(y : R)} \Rightarrow T\right)\right]\!\right]_{\mathsf{B}} \\ &\quad := \left[\!\left[\Pi[x \mapsto_{\mathsf{E}} \overline{(y : R)}/c\ \overline{y}].\ P[x \mapsto_{\mathsf{P}} c\ \overline{y}] = T[x \mapsto c\ \overline{y}]\right]\!\right]_{\mathsf{T}}\end{aligned}$$

Equations 7.2 and 7.3 would change accordingly.

This has the effect of replacing all references to the scrutinised variable in the leaves of the case tree with their discovered value.

Consequences and erasure I choose to disallow such references entirely – it is easy to reason about and simple to explain ("you cannot refer to such variables").

If we choose to allow references to inspected pattern variables, we need to choose which operational semantics they should have. This might bring potential surprise to programmers who might be expecting a different choice.

This is especially important because the two approaches I listed above differ significantly from the point of view of erasure. Consider the following pattern clause coming from a definition of $\mathsf{vlen} : (n : \mathbb{N}) \to \mathsf{Vect}\ n\ a \to \mathbb{N}$.

$$\mathsf{vlen}\ n@[\mathsf{Z}]\ \mathsf{Nil} = n$$

In this pattern clause, we bind n using as-patterns to the expression that turns out to be forced to Z.

Using the "direct reference" semantics, this function simply returns its first (explicit) argument, which means that this argument is necessarily used and cannot be erased.

Using the "substitution" semantics, the RHS would be rewritten to Z, which does not use the first argument of vlen, which can then be erased.

If we can evaluate case trees directly, we need to align the semantics of this translation to the actual semantics that we define for evaluation of case trees.

7.2.3.4 Impossible branches

Goguen, McBride and McKinna [GMM06] show that case trees need to contain explicit nodes standing for impossible cases if we want to keep checking decidable.

Such nodes would translate to **impossible** pattern clauses, as found in the definition of Idris.

7.2.3.5 Syntax sugar

Let us extend the syntax for constructor branches with an arbitrary (possibly zero) number of equalities of the form $| \; x' = R'$.

$$\begin{aligned} B \;::=\;& c \; \overline{(x : R)} \; \overline{| \; x' = R'} \Rightarrow T \\ |\;& [R] \Rightarrow T \end{aligned}$$

Extended case branches desugar into the original syntax using the following rule.

$$\begin{aligned} & c \; \overline{(x : R)} \;\; | \; x'_1 = R'_1 \ldots | \; x'_n = R'_n \Rightarrow T \\ & \qquad\qquad\qquad \Downarrow \\ & c \; \overline{(x : R)} \Rightarrow \textbf{case } x'_1 \textbf{ of} \\ & \qquad\qquad [R'_1] \Rightarrow \cdots \textbf{case } x'_n \textbf{ of} \\ & \qquad\qquad\qquad\qquad [R'_n] \Rightarrow T \end{aligned}$$

We could interpret these equalities as explicit substitutions coming from unification.

7.2.3.6 Example

Let us translate the following case tree. This time, I will elaborate the function fully, including the type argument a.

$$\begin{aligned} & \mathsf{vlen} : (a : \mathsf{Type}) \to (n : \mathbb{N}) \to \mathsf{Vect}\ n\ a \to \mathbb{N} \\ & \mathsf{vlen} = \lambda(a : \mathsf{Type})(n : \mathbb{N})(xs : \mathsf{Vect}\ n\ a). \\ & \quad \textbf{case } xs \textbf{ of} \\ & \qquad \mathsf{Nil}\ (a' : \mathsf{Type}) \\ & \qquad\quad | \; n = \mathsf{Z} \\ & \qquad\quad | \; a' = a \\ & \qquad\quad \Rightarrow \mathsf{Z} \\ & \qquad (::)\ (a' : \mathsf{Type})\ (n' : \mathbb{N})\ (x : a)\ (xs' : \mathsf{Vect}\ n'\ a) \\ & \qquad\quad | \; a' = a \\ & \qquad\quad \Rightarrow \textbf{case } n \textbf{ of} \\ & \qquad\qquad\quad \mathsf{S}\ (n'' : \mathbb{N}) \\ & \qquad\qquad\qquad | \; n' = n'' \\ & \qquad\qquad\qquad \Rightarrow \mathsf{S}\ (\mathsf{vlen}\ a\ n''\ xs') \end{aligned}$$

After desugaring, we can run the translation procedure. Some of the intermediate states of translation are outlined in the comments in Figure 7.1. We finally obtain the

```
vlen : (a : Type) → (n : ℕ) → Vect n a → ℕ
vlen = λ(a : Type)(n : ℕ)(xs : Vect n a).
  — ⟦(a : Type)(n : ℕ)(xs : Vect n a). ⌊vlen⌋ a n xs = ...⟧_T
  case xs of
    Nil (a′ : Type)
      | n = Z
      | a′ = a
      — ⟦(a : Type). ⌊vlen⌋ a [Z] (Nil [a]) = Z⟧_T
      ⇒ Z
    (::) (a′ : Type) (n′ : ℕ) (x : a) (xs′ : Vect n′ a)
      | a′ = a
      — ⟦ (a : Type)(n : ℕ)(n′ : ℕ)(x : a)(xs′ : Vect n′ a).
        ⌊vlen⌋ a n ((::) [a] n′ x xs′) = ... ⟧_T
      ⇒ case n of
          S (n″ : ℕ)
            | n′ = n″
            — ⟦ (a : Type)(n″ : ℕ)(x : a)(xs′ : Vect n″ a).
              ⌊vlen⌋ a (⌈S⌉ n″) ((::) [a] [n″] x xs′) = S (vlen a n″ xs′) ⟧_T
            ⇒ S (vlen a n″ xs′)
```

FIGURE 7.1: Selected intermediate states of translation of the case tree to pattern clauses.

following sequence of pattern clauses.

```
vlen : (a : Type) → (n : ℕ) → Vect n a → ℕ
vlen a     [Z]          (Nil [a])          = Z
vlen a (⌈S⌉ n″) ((::) [a] [n″] x xs′) = S (vlen a n″ xs′)
```

Note that the length index in the recursive call, n'', is projected out of the length argument, rather than the vector constructor, exactly as specified in the original case tree.

Also note that the length index [Z] in the first clause is a forced pattern, while ⌈S⌉ in the second clause is a forced constructor. We could make Z a forced constructor, too, if we matched on it with a single-branch case tree. For illustration, I chose it to be a forced pattern.

7.2.4 Type checking case trees

When type checking case trees, we cannot simply check that the pattern in each branch has the same type as the scrutinee because further matches may be necessary to provide missing evidence.

Chapter 5 describes how to type check pattern matching clauses. Since we can convert case trees to pattern clauses, we can check these instead of their corresponding case trees. However, more theoretical development is necessary to provide formal assurances that this approach is sound.

7.2.5 Summary

Case trees are an alternative to pattern matching clauses as a pattern matching facility of a language.

Case trees are more explicit and more operational and they would be a good component of a core language (Section 7.2.1).

We can *check* case trees without unification if they contain enough evidence of their type-correctness. Normally, the *elaborator* would perform unification and fill in the evidence so that the programmer does not have to do it. The evidence consists of the following.

- Explicit types in all binders.
- Explicit equalities in case branches (Section 7.2.3.5) or the equivalent case splits.

Having the above, we can check case trees by converting them to pattern clauses.

Equivalence of case trees and pattern clauses Because there are procedures to translate pattern clauses to case trees [Aug85; Wad87a] and back (Section 7.2.3), both representations seem to be equivalent – if we extend pattern clauses with a sequence of pattern variable names to represent the order of inspection (and explicit marking of impossible cases), both representations should be convertible back and forth losslessly.

In the absence of an explicit order of inspection in pattern clauses, forced patterns and forced constructors assert that the corresponding case inspections should have exactly one branch, which guides the choice of inspection ordering in case tree elaboration.

Thus forced patterns and forced constructors contain just enough information to recover an inspection ordering that conforms to the desired erasure pattern but do not determine it completely.

Uniqueness of type annotations In the trees-to-clauses translation, as stated above, some pattern variable binders do not survive the translation process and are substituted for by forced patterns. These binders can be annotated with arbitrary (even non-sensical) types without changing the result of the translation.

Translation from pattern clauses to case trees then cannot fill in the types of these binders (syntactically) since this information is not contained in pattern clauses. In order to achieve complete equivalence, we would have to address this, possibly by requiring explicit type annotations for forced patterns in pattern clauses.

Soundness Further work should establish the soundness of this approach formally.

7.3 Erasure polymorphism

The calculus $\mathsf{TT}_\star$, as presented so far, has a disadvantage: each function and each data constructor has a fixed erasure pattern. While the programmer can leave annotation

up to erasure inference, once annotations are inferred, they must be the same across the whole program for any given name.

It would be useful to have a mechanism that would allow us to use the same entity with different erasure patterns in different contexts, while still making sure that the program is consistent.

7.3.1 Erasure-polymorphic functions

Consider the program in Figure 7.2.

```
let
  apply : ((x : ℕ) → ℕ) → (y : ℕ) → ℕ
  apply f y = f y

  const42 : (x : ℕ) → ℕ
  const42 x = 42

  id : (x : ℕ) → ℕ
  id x = x

  expensive : ℕ
  expensive = ...
in
  T
```

FIGURE 7.2: Example of a program where erasure polymorphism of apply would be useful.

For T = (id 3 + const42 expensive), erasure inference will correctly notice that const42 does not use its argument, marking the expensive argument as erased.

However, if T = (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 type of apply that would lead to erasure of expensive in this program while keeping it consistent, even if we annotate it manually. Indeed, the function apply could have one of the following two types:

- apply : $((x :_{\mathsf{E}} \mathbb{N}) \to \mathbb{N}) \to (y :_{\mathsf{E}} \mathbb{N}) \to \mathbb{N}$
- apply : $((x :_{\mathsf{R}} \mathbb{N}) \to \mathbb{N}) \to (y :_{\mathsf{R}} \mathbb{N}) \to \mathbb{N}$

If we pick the former, apply id 3 will be inconsistent because the value 3 *is* used there. If we pick the latter, the system will fail to recognise that the argument expensive is unused in (apply const42 expensive). The inference algorithm presented in Chapter 6 will of course pick the safe/correct alternative.

Not erasing expensive is a problem that we would like to solve. Besides unnecessary evaluation (depending on the normalisation strategy), non-erasure of expensive might also (transitively) block erasure in other parts of the program.

7.3.1.1 Overview

The simplest way to achieve erasure polymorphism of any given definition is to make a copy of it every time it is referenced in the program. This works but is a bit wasteful.

Below, I describe the approach I took in my implementation, which consists of the following steps.

Inference During erasure inference, we duplicate only the type signatures and the constraints belonging to erasure-polymorphic definitions, which is sufficient to assign concrete erasure annotations to the whole program.

Specialisation For every definition, we create a copy for every *unique* erasure variant of it referenced from within the program.

This approach is then further discussed in Section 7.3.1.9 below.

7.3.1.2 Syntax

We extend the syntax of $\mathsf{TT}_\star$ terms with an extra expression form

$$\{n : term\},$$

where $\{n : \tau\}$ is a reference to an *erasure instance* of n with type τ. Here, we exploit the fact that τ contains erasure annotations, thus specifying which instance we need. In typical programs, the type is filled in by the elaborator, with erasure annotations usually left undefined, although the programmer can provide the type – and even the erasure annotations – partially or fully.

7.3.1.3 Inference

The erasure inference stage, which analyses the program in its $\mathsf{TT}^{\mathsf{evar}}_\star$ form, changes only slightly. We add an inference rule for the newly introduced syntactic form:

$$\frac{\begin{array}{c}(n :_s \sigma = b) \in \Gamma \qquad \Gamma \vdash \textsc{Def}(n :_s \sigma = b) \mid \Delta \\ \Gamma \vdash \tau :_{\mathsf{E}} \mathsf{Type} \mid \Sigma \qquad (s', \sigma', \Delta') := \textsc{Freshen}(s, \sigma, \Delta) \qquad \Gamma \vdash \sigma' \approx \tau \mid \Psi\end{array}}{\Gamma \vdash \{n : \tau\} :_G \tau \mid \Sigma \cup \Delta' \cup \Psi \cup (G \to s')}\ \textsc{EInst}$$

The (meta-)function Freshen takes an erasure annotation, a type, and a set of constraints, and renumbers all evars within the triple so that multiple occurrences of the same evar get the same new number but that number does not occur anywhere else in the original program.

Although the "freshening" operation corresponds to duplication of the type signature of the referenced value as if the referenced value was declared repeatedly in the source code, its body is not duplicated at this stage – all we need to know for erasure inference is the erasure annotation s', the type σ' with evars contained within, and the relationships between the evars, captured by the set of constraints Δ'.

The type of definition n is stored in the environment Γ. For erasure instantiation, it would be useful to store also the set of constraints for every definition in the environment to avoid recomputing it in every invocation of EInst. The syntax of definitions would thus be changed to $(n :^{\Delta}_{s} \sigma = b)$, where Δ is the associated set of constraints, but I do not address this formally here.

7.3.1.4 Solving erasure constraints

The resulting set of erasure constraints is solved exactly the same as without erasure polymorphism.

7.3.1.5 Annotation

The erasure annotation step does not change, either, and by substituting definite erasure flags R and E instead of evars, it translates the program from $\mathsf{TT}^{\mathsf{evar}}_{\star}$ to $\mathsf{TT}^{\mathsf{RE}}_{\star}$.

7.3.1.6 Specialisation

At this stage, we create erasure-specialised copies of *bodies* of erasure-polymorphic functions, one for each unique erasure pattern, and we give them fresh names. All erasure instance references will then be changed to ordinary named references to the the appropriate copy, according to their requested erasure pattern.

Definition 7.3 (Erasure pattern)**.** The erasure pattern of a type τ is the sequence of all erasure annotations occurring in τ, in some order fixed in advance.

The erasure pattern of a function or a definition is the erasure pattern of its type.

If π is an erasure pattern (i.e. a sequence of erasure annotations), then π^j stands for the j-th annotation in the sequence.

In order to specialise a program, we need to recurse simultaneously on P_{evar}, the program before it was annotated, and P_{RE}, the program after annotation. P_{RE} is the program being specialised, while P_{evar} provides "raw material": unspecialised function definitions. Specialisation produces a program in $\mathsf{TT}^{\mathsf{evar}}_{\star}$.

Figure 7.3 shows the specialisation procedure. Since P_{evar} and P_{RE} are identical up to erasure annotations, it is sufficient to define specialisation only for pairs of terms that are equal up to erasure annotations. The only interesting rules are SpecInst and SpecLet.

SpecInst A reference to an erasure instance $\{n : \tau\}$ specialises to n_π, where π is the erasure pattern of τ. The name n_π is a specific fresh name: we want to map

$$\frac{}{\left[\frac{\mathsf{Type}}{\mathsf{Type}}\right] = \mathsf{Type}}\ \textsc{SpecType} \qquad \frac{}{\left[\frac{n}{n}\right] = n}\ \textsc{SpecRef}$$

$$\frac{\left[\frac{\sigma_{\mathsf{evar}}}{\sigma_{\mathsf{RE}}}\right] = \sigma \qquad \left[\frac{T_{\mathsf{evar}}}{T_{\mathsf{RE}}}\right] = T}{\left[\frac{\lambda n :_{s_{\mathsf{evar}}} \sigma_{\mathsf{evar}}.\ T_{\mathsf{evar}}}{\lambda n :_{s_{\mathsf{RE}}} \sigma_{\mathsf{RE}}.\ T_{\mathsf{RE}}}\right] = \lambda n :_{s_{\mathsf{RE}}} \sigma.\ T}\ \textsc{SpecLam}$$

$$\frac{\left[\frac{\sigma_{\mathsf{evar}}}{\sigma_{\mathsf{RE}}}\right] = \sigma \qquad \left[\frac{T_{\mathsf{evar}}}{T_{\mathsf{RE}}}\right] = T}{\left[\frac{(n :_{s_{\mathsf{evar}}} \sigma_{\mathsf{evar}}) \to T_{\mathsf{evar}}}{(n :_{s_{\mathsf{RE}}} \sigma_{\mathsf{RE}}) \to T_{\mathsf{RE}}}\right] = (n :_{s_{\mathsf{RE}}} \sigma) \to T}\ \textsc{SpecPi}$$

$$\frac{\left[\frac{F_{\mathsf{evar}}}{F_{\mathsf{RE}}}\right] = F \qquad \left[\frac{X_{\mathsf{evar}}}{F_{\mathsf{RE}}}\right] = X}{\left[\frac{F_{\mathsf{evar}}\ \widehat{s_{\mathsf{evar}}}\ X_{\mathsf{evar}}}{F_{\mathsf{RE}}\ \widehat{s_{\mathsf{RE}}}\ X_{\mathsf{RE}}}\right] = F\ \widehat{s_{\mathsf{RE}}}\ X}\ \textsc{SpecApp}$$

$$\frac{\pi := \textsc{ErasurePattern}(\tau_{\mathsf{RE}})}{\left[\frac{\{n : \tau_{\mathsf{evar}}\}}{\{n : \tau_{\mathsf{RE}}\}}\right] = n_\pi}\ \textsc{SpecInst}$$

$$\frac{\left[\frac{\sigma_{\mathsf{evar}}}{\sigma_{\mathsf{RE}}}\right] = \sigma \quad \left[\frac{b_{\mathsf{evar}}}{b_{\mathsf{RE}}}\right] = b \quad \left[\frac{T_{\mathsf{evar}}}{T_{\mathsf{RE}}}\right] = T \quad \{\pi_i\}_{i=1}^{k} := \left\{\pi \mid n_\pi \in \mathrm{FV}(T)\right\}}{\left[\frac{\mathbf{let}\ n :_{s_{\mathsf{evar}}} \sigma_{\mathsf{evar}} = b_{\mathsf{evar}}\ \mathbf{in}\ T_{\mathsf{evar}}}{\mathbf{let}\ n :_{s_{\mathsf{RE}}} \sigma_{\mathsf{RE}} = b_{\mathsf{RE}}\ \mathbf{in}\ T_{\mathsf{RE}}}\right]}\ \textsc{SpecLet}$$

$$\begin{aligned}
&= \mathbf{let}\ n :_{s_{\mathsf{RE}}} \sigma = b\ \mathbf{in} \\
&\quad \mathbf{let}\ \textsc{Instantiate}_{\pi_1}(n :_{s_{\mathsf{evar}}} \sigma_{\mathsf{evar}} = b_{\mathsf{evar}})\ \mathbf{in} \\
&\quad\ \ldots \\
&\quad\ \ \mathbf{let}\ \textsc{Instantiate}_{\pi_k}(n :_{s_{\mathsf{evar}}} \sigma_{\mathsf{evar}} = b_{\mathsf{evar}})\ \mathbf{in} \\
&\quad\ \ \ T
\end{aligned}$$

Figure 7.3: Erasure specialisation

$$r_{[\rho\mapsto\pi]} = \begin{cases} \mathsf{R} & \text{if } r = \mathsf{R} \\ \mathsf{E} & \text{if } r = \mathsf{E} \\ \pi^j & \text{if } r = i \text{ and } \exists j.\ \rho^j = i \\ \textsc{Fresh}(i) & \text{if } r = i \text{ and } \nexists j.\ \rho^j = i \end{cases}$$

$$\frac{\rho := \textsc{ErasurePattern}(\sigma)}{\textsc{Instantiate}_\pi(n :_s \sigma = b) = (n_\pi :_s \sigma = b)_{[\rho\mapsto\pi]}}\ \textsc{InstDef}$$

Figure 7.4: Instantiation of definitions

identical erasure patterns of the same definition n to the same name but we assume that this name does not clash with anything else.

SpecLet In a let binding of name n, we must specialise the definition of n for all its erasure patterns that occur in the (already specialised!) body T. We assume that these patterns form an (arbitrarily ordered) sequence $\{\pi_i\}_{i=1}^k$.

For each i, we *instantiate* the definition $(n :_{s_{\mathsf{evar}}} = b_{\mathsf{evar}})$ according to the given erasure pattern π_i.

Observation 7.1. In SpecInst, π is the erasure pattern of τ_{RE}, which does not contain evars. Thus π is a sequence of only R and E and does not contain evars.

Instantiation of definitions Instantiation of definitions is shown in Figure 7.4. Rule InstDef defines the function $\textsc{Instantiate}_\pi$, which specialises the given definition d to $d_{[\rho\mapsto\pi]}$, changing its erasure pattern from ρ to π.

For a definition d, the operation $d_{[\rho\mapsto\pi]}$ is defined as the replacement of each erasure annotation r in the definition with $r_{[\rho\mapsto\pi]}$, where $r_{[\rho\mapsto\pi]}$ is defined in Figure 7.4.

If we cache constraints in the environment, we have to keep the set of constraints $\Delta_{[\rho\mapsto\pi]}$ in the specialised definition. Although the type signature no longer contains evars and thus does not need a set of constraints to relate them, the *body* of the specialised definition might.

Generating fresh evars In the definition of $r_{[\rho\mapsto\pi]}$, the (meta-) function $\textsc{Fresh}(i)$ generates a fresh evar that is globally unique but tied to i so that other occurrences of evar i are replaced with the same fresh evar.

Since π contains only R and E but no evars, this is the only place where evars are introduced in the specialised program.

Uniqueness of instantiation In the third branch of the definition of $r_{[\rho\mapsto\pi]}$, we replace evar i with π^j if $\rho^j = i$. The choice of j might not be unique if evar i is repeated in the erasure pattern ρ – but that cannot happen because evars in terms are uniquely numbered by the mapping from $\mathsf{TT}^\bullet_\star$ to $\mathsf{TT}^{\mathsf{evar}}_\star$.

Observation 7.2. The newly created specialisations do not contain evars in their types.

Proof. All erasure annotations in the type of an unspecialised definition are contained in ρ, by definition of ρ in Figure 7.4. Therefore the fourth, fresh-evar-generating branch of the definition of $r_{[\rho \mapsto \pi]}$ is never applicable. □

7.3.1.7 Iteration

After annotation and before specialisation, the program was expressed in $\mathsf{TT}^{\mathsf{RE}}_{\star}$ and all erasure annotations were either R or E. However, the specialisation step produces programs in $\mathsf{TT}^{\mathsf{evar}}_{\star}$ – it may introduce new evars by copying (and freshening) the bodies of erasure-polymorphic functions. If after specialisation, there are still evars left in the program, we need to repeat the inference-solving-annotation-specialisation sequence, until there are no evars left, in which case we have a $\mathsf{TT}^{\mathsf{RE}}_{\star}$ program.

Each iteration of inference, annotation, and specialisation may create several new copies of **let**-bound definitions and it may not be obvious that this process terminates. However, we can observe the following.

- Only definitions with evars in their types may be specialised, and specialisations do not contain evars in their types (Observation 7.2) so they cannot generate new specialised definitions.
- The number of different specialisations of a function is bounded by 2^n, where n is the number of evars in its type (Observation 7.1).
- If a specialisation step does not create a new instance, there resulting program can be expressed in $\mathsf{TT}^{\mathsf{RE}}_{\star}$ and does not contain evars. In such a case, the iteration terminates.

Therefore, the total number of all possible erasure specialisations in the program is bounded because the set of specialisable functions is fixed in advance and each specialisable function has a bounded number of specialisations. Each iteration strictly increases the number of specialisations present in the program, and the iteration will therefore eventually terminate.

The output of this stage is a program in $\mathsf{TT}^{\mathsf{RE}}_{\star}$.

7.3.1.8 Checking

After specialisation, the program is an ordinary $\mathsf{TT}^{\mathsf{RE}}_{\star}$ program using no erasure polymorphism since all polymorphism has been translated into use of specialised erasure instances. We can just check and further compile this program as any other $\mathsf{TT}^{\mathsf{RE}}_{\star}$ program.

7.3.1.9 Discussion

Erasure constraints as part of type signatures Erasure polymorphism suggests that erasure constraints are actually a part of a type signature[3]. The type alone may

[3]This is further supported by the usefulness of storing the set of constraints of a definition together with its type in the environment.

contain yet undefined erasure annotations represented as evars. Therefore a type with evars needs to come with a set of constraints that relates the evars in the type and describes which concrete valuations of them are erasure-consistent with the body of the definition.

For example, a reasonable type signature for apply would be as follows[4].

$$\begin{array}{l}\mathsf{apply} :_{\mathsf{R}} \{a :_{\mathsf{E}} \mathsf{Type}\} \to \{b :_{\mathsf{E}} \mathsf{Type}\} \to (f :_{\mathsf{R}} (x :_r a) \to b) \to (y :_s a) \to b \mid r \leftrightarrow t \cup t \to s \\ \mathsf{apply}\ f\ y = f\ \widehat{\ t}\ y \quad \textit{— pattern variable bindings omitted}\end{array} \tag{7.4}$$

The above type signature can be equivalently represented as a set of three signatures that are erasure-consistent with the definition of apply.

1. $\mathsf{apply} :_{\mathsf{R}} \{a :_{\mathsf{E}} \mathsf{Type}\} \to \{b :_{\mathsf{E}} \mathsf{Type}\} \to (f :_{\mathsf{R}} (x :_{\mathsf{E}} a) \to b) \to (y :_{\mathsf{E}} a) \to b$
2. $\mathsf{apply} :_{\mathsf{R}} \{a :_{\mathsf{E}} \mathsf{Type}\} \to \{b :_{\mathsf{E}} \mathsf{Type}\} \to (f :_{\mathsf{R}} (x :_{\mathsf{E}} a) \to b) \to (y :_{\mathsf{R}} a) \to b$
3. $\mathsf{apply} :_{\mathsf{R}} \{a :_{\mathsf{E}} \mathsf{Type}\} \to \{b :_{\mathsf{E}} \mathsf{Type}\} \to (f :_{\mathsf{R}} (x :_{\mathsf{R}} a) \to b) \to (y :_{\mathsf{R}} a) \to b$

Thus in the same way as the (type-) polymorphic function $\mathsf{id} : a \to a$ has a type signature that mandates that the input and output type of id must be equal but otherwise arbitrary, the type signature of apply mandates that y must not be erased if x isn't.

Runtime interfaces Each of the above three variants has a different interface at runtime: the first one erases to the identity function, the second one erases to the constant function, and only the third one erases to a function that we would normally call apply.

This means that we cannot have one runtime function that would combine the functionality of all three – we need to create specialised copies. It also means that we cannot generate code for *calling* erasure-polymorphic functions until we have decided on the particular erasure pattern to use in the call.

Separate compilation The good news is that the interface of a function is still fully determined by its type signature, which, as we have established above, includes the corresponding erasure constraints. Even if the function has not been specialised yet, we can still deduce the type and thus the runtime interface of any specialisation of it.

When performing separate compilation by modules, we can produce all (consistent) specialisations in advance (usually not many, see below) and then export the polymorphic type signature together with the names and erasure patterns of its specialisations.

[4]Erasure inference given in Chapter 6 infers a slightly more general erasure pattern – the erasability of arguments a and b would be represented by unconstrained evars and the erasability of the whole definition of apply would depend on whether the function is used anywhere in a runtime context.

We could also defer code generation for erasure-polymorphic functions by exporting unspecialised definitions together with their polymorphic type signatures, relying on users to instantiate them with specific erasure patterns.

Finally, we could also take a middle-ground approach – specialise in advance only if the number of specialisations is small, or specialise only for the erasure patterns likely to be used, including the unspecialised definition in case the caller needs an erasure pattern that has not been provided.

Constraint reduction If we are interested only in the *interface* of a function, we are interested only in the evars in its type. We could therefore implement *constraint reduction* that removes evars that do not appear in the type.

For example, in the above example with apply (Listing 7.4), the evar t does not appear in the type and the constraint set $\{r \leftrightarrow t, t \rightarrow s\}$ could be reduced to $\{r \rightarrow s\}$ without changing the meaning of the type signature.

Therefore, if a module exports a polymorphic type signature, it may be beneficial to reduce the set of constraints in this way before exporting it.

For *instantiation* of an erasure-polymorphic function however, we need a set of constraints that addresses also the evars found in the body. We therefore cannot reduce any *evars* away but depending on the representation of constraint sets in the implementation, there might be space for optimisation without removing evars.

Alternative representations of constraint sets Besides the presented representation of constraint sets as a collection of Horn clauses, there are other representations that might be useful in different cases.

My implementation represents a constraint set as a finite map from finite sets of erasure annotations to finite sets of erasure annotations. Instead of the form $G \rightarrow r$, I use the form $G \rightarrow R$ where R is the set of all annotations r implied by G. Unit propagation is easy because finding the set R belonging to $G = \emptyset$ is trivial. This representation also eliminates duplication of constraints.

Another possible representation of constraints is extensional. If there are many constraints but only a few consistent erasure patterns, it might be easier to simply list all consistent erasure patterns. Instead of a logic program, we would list all its models. This can also be seen as expressing the constraints in DNF.

Erasure-polymorphic recursion For simplicity of presentation, I tacitly ignored erasure-polymorphic recursion in Section 7.3.1.7.

If a program contains erasure-polymorphic recursion, we may need to iterate specialisation several times but we also need a way to keep the *unspecialised* definitions in their $\mathsf{TT}^{\mathsf{evar}}_{\star}$ forms. These are *not* preserved in the program by the specialisation procedure presented in Section 7.3.1.7.

A related problem is that different specialisations of a single function might end up being mutually recursive.

My implementation mostly ignores this issue and does not keep the original function in its fully unspecialised form, leading to potentially reduced polymorphism in erasure-polymorphic recursion. It also rejects any program that would require mutual recursion.

A more thorough implementation would deal with both issues properly, perhaps using the ideas introduced in Section 7.3.1.10, especially the approach that does not require further extensions to $\mathsf{TT}_\star$.

Extra syntax For erasure polymorphism of functions, we introduced extra syntax – all erasure-polymorphic functions must now be referenced using the notation $\{f : \tau\}$. This may look burdensome but this notation can be easily filled in by the elaborator.

Specialisation removes all erasure-polymorphic references.

Implementation of specialisation To reduce its time complexity, my implementation of specialisation (Figure 7.3) is a function that returns both the specialised code and the set of erasure instances referenced in it. This allows an efficient implementation of SpecLet.

Output of specialisation is monomorphic Erasure specialisation translates erasure polymorphism into erasure monomorphism. This is very good because we can support erasure polymorphism without extending our *core* core language, $\mathsf{TT}_\star^{\mathsf{RE}}$.

Erasure specialisation is therefore a form of elaboration, just like erasure inference.

Bound on number of erasure instances I claim that the number of (useful) erasure instances of an erasure-polymorphic function is not only bounded by 2^n, where n is the number of evars in its type, but further bounded by 2^m, where m is the number of evars on binders *in positive positions* in the type.

Note that for first-order functions, whose types do not contain binders in positive positions, $m = 0$.

This claim stems from the following observations.

- In an erasure-consistent program, dependencies between erasure annotations respect data dependencies: if there is data flow from a to b, then a must be retained at least as much as b.
- When looking at a single function, we can divide the binders in its type into *inputs* (binders in negative positions) and *outputs* (binders in positive positions).
- For each function, data flows from its inputs to its outputs.
- Data flow is *monotonic*: removal of data flow between a and b will not *create* more data flow elsewhere; it can only (transitively) *remove* data flow. This corresponds to the absence of negation in erasure constraints.

The above means that usage of the data flowing in through the inputs of a function is *determined* by usage of the data flowing out through outputs of the function, and by the plumbing – the body of the function.

For example, in the case of apply in Listing 7.4, y is bound in a negative position and is therefore an input, while x is bound in a positive position and is therefore an output. This agrees with the set of constraints $\{r \to t, t \to s\}$, which illustrates that the (minimal) retention of s is determined by the retention of r.

Therefore, the two useful specialisations of apply are those where $r = \mathsf{E}, s = \mathsf{E}$ or $r = \mathsf{R}, s = \mathsf{R}$.

We *could* use the erasure pattern $r = \mathsf{E}, s = \mathsf{R}$ but intuitively, that wastes the opportunity to erase y. This can be made more formal.

Definition 7.4 (Tightness of erasure patterns)**.** Let L be the (complete) lattice $\mathsf{E} < \mathsf{R}$, as defined in Section 6.1.2. Let N be the set of names bound in the type of a definition. Let $p : \{+, -\}^N$ represent the *polarity* of these names, where "+" stands for positive positions (outputs) and "−" stands for negative positions (inputs). This splits N into $N_+ := \{n \in N \mid p^n = +\}$ and $N_- := \{n \in N \mid p^n = -\}$. Let $\pi, \rho : L^N$ be two erasure patterns of the definition.

I will write π^n instead of $\pi(n)$ in order to stay consistent with the notation of erasure patterns introduced previously.

Then we can compare the *looseness* and *tightness* of π and ρ using the relation $\preceq$.

$$\pi \preceq \rho := \bigwedge_{n \in N} \begin{cases} \pi^n \leq \rho^n & \text{if } n \in N_- \\ \pi^n \geq \rho^n & \text{if } n \in N_+ \end{cases}$$

We say that π is *tighter* than ρ if the above holds strictly: $\pi \prec \rho$. We can also say that ρ is *looser* than π.

Claim 7.1 (Loosening and consistency)**.** *If $\pi \preceq \rho$ are erasure patterns of a definition and π is consistent, then ρ is also consistent. Consistent erasure patterns thus form an upper set in $(L^N, \preceq)$.*

As an example, the erasure patterns of apply form the structure shown in Figure 7.5. The bottom pattern is too tight and thus inconsistent (marked by ✗). I am going to claim that the top pattern is, conversely, too loose, even if consistent (marked by ✓).

Definition 7.5 (Tight and loose erasure patterns)**.** For a given definition, its erasure pattern is *tight* if there is no tighter erasure pattern for that definition that would be erasure consistent. Otherwise, the erasure pattern is *loose*.

Claim 7.2 (Loose erasure patterns are wasteful)**.** *For any definition, if its erasure pattern π is consistent but loose, there is a pattern $\rho \prec \pi$ that is:*

- *(strictly) tighter than π;*
- *consistent;*

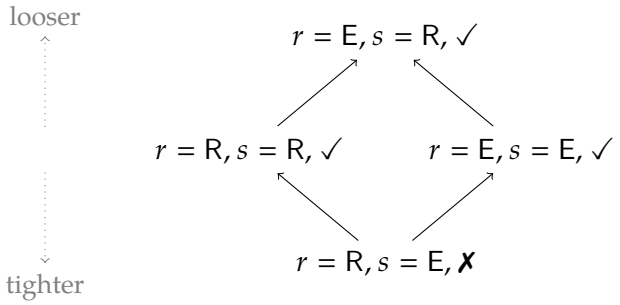


FIGURE 7.5: Erasure patterns of apply by tightness

- *agrees with π on N_+.*

In other words, we can produce the same output consuming strictly fewer resources.

Claim 7.3. *For any definition, if both π and ρ are tight and consistent erasure patterns and they agree on N_+, then $\pi = \rho$.*

Claim 7.4. *Erasure inference presented in this dissertation produces tight and consistent erasure patterns.*

Therefore, the set of up to 2^n erasure patterns, where $n \leq |N|$ is the number of evars in the type of any given definition, collapses by tightening of loose and wasteful patterns into a set of 2^m erasure patterns, where $m = |N_+|$ is the number of positively bound output names.

Erasure inference will pick one of the 2^m erasure patterns and it is therefore usually sufficient to generate just these instances. However, the programmer can still request a loose erasure pattern using explicit annotations and the compiler should be able to deal with this case by using a possibly less efficient fallback strategy, such as instantiation on demand.

Richer lattices of erasure annotations Although this is outside of the scope of this dissertation, the principles shown here should be directly adaptable for any complete lattice of erasure annotations, not just $\mathsf{E} < \mathsf{R}$ (Section 9.2.1.11).

7.3.1.10 Explicit erasure quantification

An interesting direction of research might be a calculus that would allow quantification over erasure annotations as over any other value, using a special type $\mathcal{R}$ with two values, $\mathsf{E} : \mathcal{R}$ and $\mathsf{R} : \mathcal{R}$.

While this is already implementable in $\mathsf{TT}_\star$ without any extensions (see below), it might be interesting (and possibly more efficient) to have a direct support in the programming language.

With erasure quantification, the following $\mathsf{TT}_\star$ function,

$$\begin{aligned}&\mathsf{apply} : \{a : \mathsf{Type}\} \to \{b : \mathsf{Type}\} \to (f : (x : a) \to b) \to (x : a) \to b\\ &\mathsf{apply}\ \ f\ \ x\ =\ f\ x\end{aligned}$$

would be expressed in this hypothetical calculus as follows.

$$\begin{aligned}
&\mathsf{apply}\ :\ \{a :_{\mathsf{E}} \mathsf{Type}\} \to \{b :_{\mathsf{E}} \mathsf{Type}\} \to (r :_{\mathsf{R}} \mathcal{R}) \\
&\qquad \to (f :_{\mathsf{R}} (x :_r a) \to b) \to (x :_r a) \to b \\
&\mathsf{apply}\ r\ f\ x\ =\ f\ \widehat{r}\ x
\end{aligned}$$

The above would then elaborate to the following.

$$\begin{aligned}
&\mathsf{apply}\ :\ \{a :_{\mathsf{E}} \mathsf{Type}\} \to \{b :_{\mathsf{E}} \mathsf{Type}\} \to (r :_{\mathsf{R}} \mathcal{R}) \\
&\qquad \to (f :_{\mathsf{R}} (x :_r a) \to b) \to (x :_r a) \to b \\
&\mathsf{apply}\ \mathsf{E}\ f\ x\ =\ f\ \widehat{E}\ x \\
&\mathsf{apply}\ \mathsf{R}\ f\ x\ =\ f\ \widehat{R}\ x
\end{aligned}$$

Finally, this would erase into the following function.

$$\begin{aligned}
&\mathsf{apply}\ \mathsf{E}\ f\ = f \\
&\mathsf{apply}\ \mathsf{R}\ f\ x = f\ x
\end{aligned}$$

Perhaps surprising is the annotation R in the binding $(r :_{\mathsf{R}} \mathcal{R})$. It turns out that it makes sense to make erasure annotations *retained* in erasure-polymorphic functions, as this will automatically choose the desired erasure variant.

Target for erasure polymorphism This is therefore an alternative to duplication/copying, as presented in Section 7.3.1.6, because it eliminates the problem of erasure-polymorphic recursion elaborating into mutual recursion (Section 7.3.1.9).

Note that we do not need to extend our calculus to use this approach to elaborate erasure polymorphism. Instead of using the quantified erasure variables as erasure annotations, which requires special support from the calculus, we can define a separate type function, which does not.

In the case of apply, the elaboration would look like the following.

$$\begin{aligned}
&\mathsf{applyTy} : (r :_{\mathsf{R}} \mathcal{R}) \to \mathsf{Type} \\
&\mathsf{applyTy}\ \mathsf{E}\ =\ \{a :_{\mathsf{E}} \mathsf{Type}\} \to \{b :_{\mathsf{E}} \mathsf{Type}\} \to \big(f :_{\mathsf{R}} (x :_{\mathsf{E}} a) \to b\big) \to (x :_{\mathsf{E}} a) \to b \\
&\mathsf{applyTy}\ \mathsf{R}\ =\ \{a :_{\mathsf{E}} \mathsf{Type}\} \to \{b :_{\mathsf{E}} \mathsf{Type}\} \to \big(f :_{\mathsf{R}} (x :_{\mathsf{R}} a) \to b\big) \to (x :_{\mathsf{R}} a) \to b
\end{aligned}$$

$$\begin{aligned}
&\mathsf{apply} : (r :_{\mathsf{R}} \mathcal{R}) \to \mathsf{applyTy}\ r \\
&\mathsf{apply}\ \mathsf{E}\ f\ x\ =\ f\ \widehat{\mathsf{E}}\ x \\
&\mathsf{apply}\ \mathsf{R}\ f\ x\ =\ f\ \widehat{\mathsf{R}}\ x
\end{aligned}$$

Note that the above program can be expressed in $\mathsf{TT}_\star$ already and no extensions are needed.

Finally, also note that this approach is similar to simulating mutual recursion, as shown in Section 2.2.4.3. This also means that the "switch" argument that chooses the erasure variant need not have the type $\mathcal{R}$ – if there are several variants, we can

split on a custom special-purpose ADT with just the right number of constructors (possibly indexed with erasure annotations).

7.3.1.11 Conclusions

The most important conclusions of this section are the following.

- It is useful to think about erasure constraints as part of a type signature.
- Specialisation *elaborates* erasure polymorphism into a fully monomorphic form expressed in $\mathsf{TT}^{\mathsf{RE}}_{\star}$, our core calculus.
- Erasure polymorphism does not stand in the way of separate compilation, although a bit of care needs to be taken.
- The number of useful erasure patterns (and thus instances) of an erasure-polymorphic function depends on the number of evars only at names bound *in positive positions* in the type. For first-order functions, there is only one useful erasure pattern.

7.3.2 Erasure-polymorphic type families

Consider the following program, which is inspired by a real issue that arose in the standard library of Idris after implementing erasure. The program below defines the function filter whose type guarantees that all elements that survive filtering satisfy the given predicate. The type constructor Dec implements the notion of decidable predicates (Section 2.2.2.2), and the type constructor Sigma implements dependent pairs.

```
data Sigma : (a : Type) → (a → Type) → Type where
  (,) : (x : a) → p x → Sigma a p

data All : (a → Type) → List a → Type where
  Nil_A  :  All p Nil
  (::_A) :  p x → All p xs → All p (x :: xs)

filter : (decP : (x : a) → Dec (p x)) → List a → Sigma (List a) (All p)
filter decP     Nil      = (Nil, Nil_A)
filter decP  (x :: xs) = case (decP x, filter decP xs) of
                           (Yes px,  (xs, pxs)) ⇒ (x :: xs, px ::_A pxs)
                           (No npx,  (xs, pxs)) ⇒ (xs, pxs)
```

It turns out that in the function filter, it is impossible to erase the second component of the return value – the proof that all elements of the returned list satisfy the given predicate.

This happens because we used (dependent) pairs for two different purposes. We use a (dependent) pair to return a value together with a proof. We expect the second

element to be erased. However, we use a (non-dependent) pair for simultaneous pattern matching in the case expression[5], where both elements of the pair must be retained.

Since the pair constructor $(,)$ used in both cases is the same, the only consistent erasure annotation for it is to erase nothing.

Although I have not implemented erasure polymorphism for type families in real code yet, I will sketch the idea of how I would approach it.

7.3.2.1 Intuition

The key observation is that the two different usages of the pair constructor (and its type) in the example above are *disjoint* – we could well use a different pair-like type in the case expression than in the return type of the function and the program would still typecheck.

On the other hand, if $(f : a \to \mathsf{Sigma}\ a\ p)$ and $(g : \mathsf{Sigma}\ a\ p \to b)$ and $(g \circ f)$ appears in the program, then clearly the two uses of Sigma are *not* disjoint – they certainly have to represent the same type family.

Therefore, I informally define "disjointness" using the question "can these constructors belong to different type families and still typecheck?". The inference algorithm starts by assigning a separate type family to each constructor and then use type checking to discover where typechecking would fail if the type families do not match, recording all such cases as equivalence constraints.

The equivalence classes on constructor references given by these constraints then represent the maximal granularisation of type families that still typechecks. We split/duplicate type families at this granularity.

Then we perform erasure inference on the resulting $\mathsf{TT}^{\mathsf{evar}}_{\star}$ program as usual, obtaining a program in $\mathsf{TT}^{\mathsf{RE}}_{\star}$ with only concrete erasure annotations.

Finally, if some (copies of) type families turn out to have the same erasure pattern in $\mathsf{TT}^{\mathsf{RE}}_{\star}$, we can merge them before proceeding with compilation as usual.

7.3.2.2 Disjointness inference

Disjointness inference happens *before* erasure inference and its input and output is a program expressed in $\mathsf{TT}^{\mathsf{evar}}_{\star}$.

To find out which uses of type families and their constructors are disjoint, I propose the following procedure.

1. Each reference to a type constructor or a data constructor in the program is numbered with a unique number i, changing each data constructor c to c_i and

[5]One could argue that we could use nested case expressions, pattern matching **let** or explicit projections instead of matching on a pair. That's however less convenient and it limits the way programmers can write the code. Alternatively, one can consider the function partition : $(decP : (x : a) \to \mathsf{Dec}\ (p\ x)) \to \mathsf{List}\ a \to \mathsf{Pair}\ (\mathsf{Sigma}\ (\mathsf{List}\ a)\ (\mathsf{All}\ p))\ (\mathsf{Sigma}\ (\mathsf{List}\ a)\ (\mathsf{All}\ (\mathsf{Not} \circ p)))$ instead of filter, where there is no alternative because the return type alone uses pairs in two different ways.

each type constructor T to T_i for a fresh number i. Let us call this number an *fvar* ("ef-var").

We can interpret that as creating a unique copy of its corresponding type family for each reference to a type or data constructor.

2. Type check program using constraint-generating rules. These rules have the form $i \sim j$ for fvars i and j. We need a constructor conversion rule, given as follows.

$$\frac{(c :_s \sigma = \textbf{constructor}) \in \Gamma}{\Gamma \vdash c_i \approx^{\{i \sim j\}} c_j} \textsc{ConvReflCtor}$$

Furthermore, each type signature of a data constructor $c_i : \overline{(n :_s \sigma)} \to T_j(\overline{n})$ must generate a constraint $i \sim j$. This establishes the link between a constructor and its type family.

The remaining typing rules just integrate the constraints from subterms.

The resulting constraint set contains pairs of fvars that stand for *non-disjoint* constructors.

3. Let $\sim^\star$ be the reflexive symmetric transitive closure of $\sim$. This partitions the set of all fvars F into equivalence classes $F/{\sim^\star}$.

4. All fvars in one equivalence class represent (type and data) constructors of the same type family. Therefore, there is a suitable function mapping equivalence classes to type families.

Proof sketch. We start with each constructor in its own equivalence class.

In each typing rule that generates any new constraints, the constraints are generated for constructors of the same type family.

5. Let Φ be the set of all type families in the program so far and let $f : F/{\sim^\star} \to \Phi$ assign the name of the corresponding type family to each equivalence class.

For each type family T, we create a copy T_c for each equivalence class c in the preimage $f^{-1}(T)$.

The erasure annotations in each copy need to be freshened. Because we assume that erasure inference has not been performed yet and there are no erasure constraints present in the program, we can simply generate entirely fresh evars. This is different from Section 7.3.1.3, where we needed to preserve the relationships between evars in the instantiated definition.

6. Finally, rewrite all references to type and data constructors in the program, changing c_i to $c_{[i]}$ everywhere, where $[i]$ is the equivalence class of fvar i.

The result is a program in $\mathsf{TT}^{\mathsf{evar}}_\star$.

7.3.2.3 Erasure inference

Erasure inference, possibly including elaboration of polymorphic functions (Section 7.3.1), runs after the above procedure and produces a program in $\mathsf{TT}^{\mathsf{RE}}_{\star}$.

7.3.2.4 Merging families

Erasure inference may infer exactly identical erasure patterns for different copies of a type family introduced in the disjointness inference step. Such families may be merged into one. The result is a program in $\mathsf{TT}^{\mathsf{RE}}_{\star}$, which can be checked and further compiled as usual.

7.3.2.5 Discussion

Syntax Erasure polymorphism of type families does not introduce any new syntax, unlike erasure polymorphism of functions introduced in Section 7.3.1. Therefore, erasure polymorphism of type families is not explicitly marked in code and need not be performed to elaborate the extra syntactic elements away.

Indeed, the procedure for *erasure polymorphism* of type families does not deal with erasure, nor with polymorphism at all. It is in fact a procedure for splitting type families into as many copies as possible before erasure inference, and then merging identically annotated families after erasure inference. The achieved erasure polymorphism is only a fortunate side effect.

Iteration Since erasure might break links in data flow, it may increase disjointness of constructors. This means that multiple iterations of splitting, erasure inference and merging might achieve better erasure.

Iteration works because, informally, erasure annotations do not matter in erased contexts and different copies of the same type family differ only in erasure annotations. Therefore, in erased contexts, we may freely confuse different copies of the same type family, which means that disjointness inference need not generate any constraints from terms in erased contexts.

Order of inference While erasure inference may benefit from more granular type families generated by splitting them, splitting type families *after* erasure inference does not improve erasure or performance of the compiled program in any way.

Copy-polymorphic functions There is an important problem with this scheme. Consider the following program.

```
data Pair : Type → Type → Type where
  P : a → b → Pair a b

fst : Pair a b → a
fst (P x y) = x
```

```
snd : Pair a b → b
snd (P x y) = y

const : a → b → a
const x y = x


main : ℕ
main = const (fst xy) (fst xy) + const (snd pq) (fst pq)
 where
   xy = P 3 expensive
   pq = P 4 2
   expensive = ...
```

The second field of xy is expensive and it is not used anywhere, therefore we would like to erase it. It might seem that the family Pair, together with its constructor P could be split in two because the uses of the two pairs xy and pq are entirely disjoint in terms of data flow.

However, they are not disjoint type-wise. The second arguments of const in the body of main are (fst xy) and (fst pq). Since the projection fst has one specific type for one specific copy of Pair, it follows that the copy of Pair in pq must be the same as the copy of Pair in xy.

More generally, this means that using projection functions severely interferes with this type-based erasure polymorphism for inductive type families.

A solution might be extending the polymorphism of data constructors and functions in the following sense. Currently, we annotate the names of constructors with the fvars of their target family, like in the following (contrived) example, and we do not annotate the names of functions at all.

$$\begin{aligned} \mathsf{C}_f &: \mathsf{Bool} \to (\mathsf{Pair}\ a\ b) \to \mathsf{T}_f \\ \mathsf{projBool} &: \mathsf{T}_f \to \mathsf{Bool} \end{aligned}$$

We could however annotate both constructors and functions with *all* fvars occurring in their types.

$$\begin{aligned} \mathsf{C}_{f_1 f_2 f_3} &: \mathsf{Bool}_{f_1} \to (\mathsf{Pair}_{f_2}\ a\ b) \to \mathsf{T}_{f_3} \\ \mathsf{projBool}_{f_1 f_2} &: \mathsf{T}_{f_1} \to \mathsf{Bool}_{f_2} \end{aligned}$$

This would yield an approach very similar to annotating names n with their erasure pattern π that we saw in Section 7.3.1.6. The efficiency of this approach is however an open question.

Another solution might be a smarter analysis that does not have to be perfect if it solves the most frequently occurring issues. Finally, this might be improved using a form of CFA – or some other approach.

7.4 Non-whole-program analysis

While making constraints part of a type signature (Section 7.3.1.9) allows us to run erasure inference on different modules of a larger program separately, we cannot produce a definitive erasure pattern (and thus generate machine code) until we have seen the whole program.

For example, consider a program containing, among others, the following two definitions.

```
data T : Type where
  C : (x : ℕ) → T

f : (x : ℕ) → T
f x = C x
```

Is the argument x of function f erasable? The answer is that we cannot tell until we have seen the whole program. If any function *reads* the field x from the constructor C in a runtime context, we must not erase x neither in C, nor in f. If there is no such function, we can erase both.

This happens because erasure uses *negation as failure*: only if it fails to show that a variable is necessary for runtime, it concludes that the variable is erasable. However, by reading constructor fields, any part of the program may introduce new constraints that would make the field and all its dependencies necessary for runtime in entirely unrelated parts of entirely unrelated modules.

This limitation is similar to that of 0-CFA in that it does not matter in what context *someone*, *somewhere* reads the constructor field in question, and it does not matter whether the read value could possibly have originated from f. Erasure is blocked nevertheless.

Section 7.3.2 introduced a type-based analysis that might able to help slightly by giving different names to data constructors used in obviously different contexts.

Below, I list a few ideas addressing this problem.

7.4.1 Explicit erasure patterns for data constructors

One way out would be requiring programmers to specify fixed erasure patterns for data constructors together with their definition. Then the "action at distance" of data constructors would be eliminated and each function could be analysed and compiled separately.

Of course, it would not be very satisfactory to develop an erasure method just to conclude that we are better off with manual annotations.

7.4.2 Module-restricted erasure inference

A better solution might be inferring an erasure pattern for data constructors only from the module they are contained in (or another programmer-selected set of modules).

Afterwards, the erasure pattern would be fixed and other modules that import the data type definitions would not be able to influence it. Any runtime access of an erased value would become a compile-time error (namely, erasure inconsistency).

7.4.3 Smarter negation-as-failure

Even if we keep the "action at distance" of data constructors, some variables can still be recognised as erasable very early. For example, in the function (const $x\ y = x$), the argument y is erasable, regardless of erasure patterns of any data constructors. This generally happens for erasure-monomorphic functions that do not depend on reading values from constructor fields.

We could therefore extend erasure inference to recognise such cases and mark some binders as erasable before the whole set of constraints has even been collected.

This would not solve the problem of whole-program compilation but it might reduce its cost by allowing some functions to be compiled separately ahead-of-time and leaving compilation of other functions until after the full erasure pattern has been established.

Alternatively, this method would be composable with the previous method of restricting action at distance to single modules (or other declared scopes).

7.4.3.1 Implementation of smarter negation-as-failure

We can split evars into two categories – those that depend on a constructor field and those that do not, which is decidable syntactically as follows.

Since a variable cannot be referenced outside of its scope, its scope is the only possible source of constraints of the form $G \to r$, where r is the erasure annotation of the variable. Let Δ be the set of constraints coming from the scope of a variable.

Definition 7.6 (Dependence on a constructor field)**.** Evar *i depends on a constructor field* in Δ iff at least one of the following two conditions hold.

- i is the erasure annotation of a pattern variable that occurs guarded by a constructor;
- $(G \to i) \in \Delta$ where some $g \in G$ depends on a constructor field.

Definition 7.7 (Action at distance)**.** Variable n with scope constraints Δ is *subject to action at distance* iff it is bound with erasability i where i is an evar that depends on a constructor field in Δ.

Claim 7.5. *Any variable n that is not subject to action at distance can be assigned a definite erasure annotation R or E immediately after gathering the constraints from its scope.*

Since we know that there will be no more constraints that could force n to be annotated with R, we can use the closed-world assumption even with an incomplete set of constraints.

I have not implemented this extension.

7.5 I/O and FFI

My implementation shows that erasure works well with monadic I/O and FFI and a short summary of this section would be "no surprises at all".

7.5.1 Foreign postulates

My implementation features FFI via foreign postulates. To add foreign postulates to the calculus, we can extend the production rule *body* in Figure 5.2 with another abstract body form $\textbf{foreign}_{code}$, where *code* is a string describing the foreign implementation of the postulate[6].

Foreign postulates generally behave like variables: they stand for an unknown value (Section 5.2.2.1) and, just like with variables, matching a postulate with a complex pattern should make the match stuck.

Now we can invoke foreign functions, as illustrated by the following programs.

```
let
  sayHello : (x :R ()) → () = foreign...
in
  sayHello ()
```

My Scheme code generator allows printing any value:

```
let
  printInternalRepr : (a :E Type) → (x :R a) → () = foreign...
in
  printInternalRepr ℕ 4
```

The erasure pattern of foreign functions must generally be given explicitly since erasure inference has no way to know how they are implemented and how they use their arguments.

Reduction at compile time At compile/typechecking time, foreign postulates are abstract and they do not reduce, like variables. However, the compiler assumes that – like variables – they *might* reduce later, as mentioned above.

7.5.2 Monadic I/O

The functions sayHello and printInternalRepresentation are clearly impure and we might want to manage their side effects using a monadic API.

My implementation shows that erasure works well with monadic I/O, as used in pure functional languages, and there are no surprises – and no erasure annotations are needed, except for foreign postulates, as mentioned above, and one annotation in ioWrapImpure (see below).

[6]Since my implementation compiles to native code via Scheme, *code* is a Scheme expression.

State I assume an (entirely standard) implementation of the State monad, with type constructor State : $(st : \mathsf{Type}) \to (a : \mathsf{Type}) \to \mathsf{Type}$ and operations like the following.

$$
\begin{aligned}
\mathsf{get} &: \mathsf{State}\ st\ st \\
\mathsf{return} &: a \to \mathsf{State}\ st\ a \\
(\gg\!=) &: \mathsf{State}\ st\ a \to (a \to \mathsf{State}\ st\ b) \to \mathsf{State}\ st\ b
\end{aligned}
$$

From State to IO Then the I/O monad is defined, unsurprisingly, as the state monad where the state is RealWorld, a token with no informative content, used only to obtain the desired evaluation order. [PJW93]

$$
\begin{aligned}
&\textbf{data}\ \mathsf{RealWorld} : \mathsf{Type}\ \textbf{where} \\
&\quad \mathsf{TheWorld} : \mathsf{RealWorld} \\
\\
&\mathsf{IO} : \mathsf{Type} \to \mathsf{Type} \\
&\mathsf{IO}\ a = \mathsf{State}\ \mathsf{RealWorld}\ a
\end{aligned}
$$

A real-world standard library would hide the constructor TheWorld.

Wrapper for side-effectful actions Low-level effectful actions that compute a value of type a are represented as (impure) functions with the type $(w :_{\mathsf{R}} \mathsf{RealWorld}) \to a$. We can wrap such functions as I/O actions using the following function.

$$
\begin{aligned}
&\mathsf{ioWrapImpure} : \left(f : (w :_{\mathsf{R}} \mathsf{RealWorld}) \to a\right) \to \mathsf{IO}\ a \\
&\mathsf{ioWrapImpure}\ f = \mathsf{get} \gg\!= \left(\lambda w.\ \mathsf{return}\ (f\ w)\right)
\end{aligned}
$$

Note the annotation R on w in the type. This is the only erasure annotation that we need in non-foreign code and its purpose is to declare that any function f passed to ioWrapImpure must not have its argument erased.

In other words, since functions taking a token argument (such as RealWorld) represent delayed computations, this annotation effectively asserts that f remains delayed at runtime, too.

Erasure inference then ensures that this annotation is propagated to the appropriate places, most importantly the state of the State/IO monad.

Using the wrapper Having the I/O machinery defined, we can now define new I/O actions via FFI like in the following example.

$$
\begin{aligned}
&\mathsf{printInternalRepr} : (x : a) \to \mathsf{IO}\ () \\
&\mathsf{printInternalRepr}\ x = \\
&\quad \textbf{let}\ \mathsf{printImpure} : \{a :_{\mathsf{E}} \mathsf{Type}\} \to (x :_{\mathsf{R}} a) \to () = \textbf{foreign}_{\ldots} \\
&\quad\quad \textbf{in}\ \mathsf{ioWrapImpure}\ (\lambda w.\ \mathsf{printImpure}\ x)
\end{aligned}
$$

The above assumes that the foreign function printImpure returns the low-level representation of ().

The argument to ioWrapImpure has the form $(\lambda w.\ \mathsf{print}\ x)$. The value w is ignored; the lambda is used only to delay the computation in its (impure) body and the corresponding side effects.

Separation of concerns Note that in the above definition of printInternalRepr contains explicit erasure annotations only on the foreign postulate. We cannot avoid this – erasure analysis cannot know how the represented function uses its arguments. However, except for the foreign postulate, there are no other erasure annotations.

The argument w of the lambda passed to ioWrapImpure must not be erased so that the delayed computation stays delayed at runtime. This has been stated *once*, in the type signature of ioWrapImpure, and, thanks to erasure inference, it never needs to be repeated again.

Erasure inference is also able to infer the erasure pattern of printInternalRepr.

This means that explicit annotations are kept at the absolute minimum: we have to use exactly one annotation (in the type of ioWrapImpure), plus we have to annotate every foreign postulate.

7.6 Irrelevance

Irrelevance of certain data (mostly proofs) is convenient for programming, and it would be useful to have it in $\mathsf{TT}_\star$ in some form.

7.6.1 Irrelevance of all erased values

Since all irrelevant values can be erased, some literature takes the approach of *identifying* erasability with irrelevance, such as ICC* of Barras and Bernardo [BB08] or Zombie [Sjö15]. (See also Section 3.2.1.3 for comparison with Agda.) Mishra-Linger [ML08] also considers this option, which he calls "elective[7] proof irrelevance", and describes its consequences.

Why As described in Section 2.2.4.3, irrelevant terms (unlike erased terms) can be disregarded already at the stage of type checking, which would make type checking more efficient. Furthermore, a system with a joint notion of irrelevance/erasure is arguably simpler than one with two distinct notions.

Why not Irrelevance is distinct from erasure and we may want to erase non-irrelevant values, as discussed in Section 2.1.7.

Furthermore, adding irrelevance complicates the compiler (as partly described below) and the theory, while erasure alone yields all the run-time performance gains without the invasive changes that adding irrelevance would bring.

[7] Unlike "universal proof irrelevance" that would be implemented in a system like Coq for the whole of Prop by a proof irrelevance axiom, *elective* proof irrelevance can be opted-into by the programmer only where desirable, using explicit erasure annotations. However, *all* erased values *must* be irrelevant, and the programmer cannot elect to have only some erased values irrelevant.

7.6.1.1 Making all erased terms irrelevant

Following Mishra-Linger's EPTS• [ML08], we can make erased terms irrelevant by modifying the conversion rule of $\mathsf{TT}^{\mathsf{RE}}_{\star}$ as follows.

$$\frac{\Gamma \vdash T :_r \tau \qquad \langle\Gamma\rangle \vdash \langle\tau\rangle \approx \langle\sigma\rangle \qquad \Gamma \vdash \sigma :_{\mathsf{E}} \mathsf{Type}}{\Gamma \vdash T :_r \sigma}\ \textsc{ConvIrr}$$

Since erasure preserves conversion (Corollary 5.59), rule ConvIrr is more permissive than the original rule Conv.

Erasure inference now becomes a bit more complicated because $\mathsf{TT}^{\mathsf{evar}}_{\star}$ terms can contain evars and the conversion rule does not know what to erase yet. Rule ConvIrr would thus make *erasure* influence *type*-correctness.

This agrees with the observation of Mishra-Linger [ML08, Sec. 6.4, Sec. 8.3] that erasure inference may not carry over to his calculus with irrelevance of erased values, EPTS•, although Mishra-Linger also suggests trying a non-complete approximation of the inference rules, and see if it yields a useful approach in practice.

I suggest such an approach in Section 7.6.3 below.

7.6.2 Irrelevance of some erased values

Instead of identifying erasure and irrelevance, we could extend $\mathsf{TT}_{\star}$ with irrelevance as an *erasure level* so that each binder would be marked with one of the following three symbols.

Symbol	Compile time	Run time
I	irrelevant	erased
E	relevant	erased
R	relevant	retained

At compile time, arguments of I-applications of functions would be disregarded in conversion checking. At run time, I- and E-bound values would be erased.

Three annotations above would naturally form the lattice $\mathsf{I} < \mathsf{E} < \mathsf{R}$, and therefore we could use this as an extended lattice of erasure levels instead of $\mathsf{E} < \mathsf{R}$, as described in Section 9.2.1.11.

The distinction between irrelevant and erased values now allows us to run normal erasure inference (Chapter 6), while minimising the manual annotation by annotating only the irrelevant values.

Alternatively, we can use a more complicated inference algorithm, as outlined below.

7.6.3 Inference of irrelevance

One way to extend erasure inference to support irrelevance is deferring the type equivalence checks from the conversion rule and interleaving them with solving

erasure constraints: if we learn that a certain term cannot be irrelevant, we can perform the equivalence checks within and extend the constraint set with the constraints generated by them. Fortunately, since adding more constraints cannot "make things more erased", the newly added constraints cannot invalidate the conversion checks that they arose from.

In practice, this can be realised by extending the syntax of erasure constraints to also include the form $G \to (\Gamma \vdash T \approx T')$. Then if $\left(\bigwedge G\right) > \mathsf{I}$, which means that all evars in G are non-irrelevant, we schedule a term/type equivalence check $\Gamma \vdash T \approx T' \mid \Delta$, which can possibly produce additional constraints Δ. If this check fails, then the whole term must fail to typecheck. If this check succeeds, we add the set of constraints Δ, generated from term equivalence checking, to the set of constraints for the whole program, and run the solver again, until we reach a fixed point.

Since now also part of the type checking relies on constraints, it is even more important to keep the provenance with each constraint in order to produce useful error messages.

I have implemented this inference mechanism, with $\mathsf{I} < \mathsf{E} < \mathsf{R}$, for a small PTS calculus featuring only variables, lambdas, pi, and applications.

7.7 Better error messages

The principles from Section 4.5.6 apply here as well.

Chapter 8

Related work

8.1 Direct influences

$\mathsf{TT}_\star$ builds on

- TT, the core calculus of Idris [Bra13];
- the erasure semantics presented in Mishra-Linger's dissertation [ML08].

8.1.1 TT

TT is the core calculus of Idris [Bra13].

TT supports inductive types and provides pattern matching via pattern matching clauses. $\mathsf{TT}_\star$ extends them with erasure annotations, erasure inference, and further extensions.

In TT, function- and datatype definitions are *global* – they extend TT with the corresponding constants and reduction behaviour. In $\mathsf{TT}_\star$, all definitions are **let**-bound locally.

Theorem 5.1 proves subject reduction for $\mathsf{TT}_\star$. While it has not been formally extended to TT, it is reasonable to expect that, for example by setting all erasure annotations to R, Theorem 5.1 provides a good argument for subject reduction of Idris, in particular for pattern matching with pattern clauses.

Lemma 5.50 also suggests why in TT, patterns can be checked simply as terms – if we disregard erasure annotations, the pattern typing rules of $\mathsf{TT}_\star$ coincide with the term typing rules.

8.1.1.1 Idris

Idris [Bra13] is a general purpose pure functional programming language with dependent types. Its core calculus TT (Section 8.1.1) has been used as the basis for $\mathsf{TT}_\star$ (Chapter 5). Compared to TT however, $\mathsf{TT}_\star$ supports erasure and local (**let**-bound) pattern matching definitions.

Like Idris, $\mathsf{TT}_\star$ uses pattern clauses to express pattern matching. These are compiled to case trees after typechecking (see Section 7.2.2.1).

Before the erasure method from Chapter 4 was implemented, Idris used forcing, detagging, and collapsing [BMM04] to erase unnecessarily duplicated data. This is further discussed in Section 3.2.1.1.

Currently, Idris uses the erasure method described in Chapter 4 of this dissertation. This change was merged into the main source tree in April 2014 and it runs on all programs.

8.1.1.2 Epigram

Epigram [MM04; McB05; GMM06] is a dependently typed programming language, predating Idris and also based on TT, although currently unmaintained. Epigram has pioneered novel developments in dependently typed programming, such as the **with** notation [MM04] (Section 2.2.5), the forcing, detagging, and collapsing optimisations [BMM04], or translation of dependent pattern clauses to eliminators [GMM06].

8.1.2 EPTS

The dissertation of Richard Nathan Mishra-Linger [ML08] presents a language, EPTS, with many desirable features, most notably an *extrinsic view of erasure*, where "being erased" is not viewed as an intrinsic property determined by the type of a value (unlike in Coq, for example) but as a contextual property – it depends on where the value occurs.

Erasure Pure Type Systems [ML08] are a variation of Pure Type Systems [Bar91], extended with erasure annotations. Mishra-Linger gives an erasure inference algorithm, which fills erasure annotations in unannotated programs, and an erasure transformation, which erases parts of programs annotated as erasable.

Furthermore, Mishra-Linger does not necessarily postulate equality of all erased values. Erasure is motivated operationally and it describes whether a value can affect run time operational behaviour of a program. Agda's irrelevance, on the other hand, conflates all inhabitants of a type already at the point of typechecking, which makes it less flexible. Mishra-Linger, however, also considers ICC-style [Miq01; BB08] irrelevance of erased values in his calculus EPTS$^\bullet$.

Mishra-Linger develops comprehensive theoretical links between polymorphism, parametricity, irrelevance, and erasure, and gives a very solid metatheoretic account of his system, ranging from the metatheory of EPTS, inductive types (with eliminators), erasure constraint solving, to elective proof irrelevance and squash types.

While eliminators were sufficient for Mishra-Linger to develop a lot of interesting metatheory, and there is a way to translate pattern matching to eliminators [GMM06], for a practical programming language, we might want to support inductive data types more directly. Today's programming languages generally use pattern matching and separately termination-checked recursion, which is more flexible and more operationally direct than fixed induction principles and their eliminators.

Mishra-Linger's calculi use context reset (also known as resurrection), a potentially costly operation unless implemented cleverly. $\mathsf{TT}_\star$ instead allows "downcasting" variables in the Ref checking rule and uses sets of guards in the inference rules.

Mishra-Linger's approach to erasure forms an excellent foundation for further development. Chapter 5 of this dissertation adds (full dependent) pattern matching, and Chapter 7 describes further extensions, like erasure polymorphism, case trees, and a different form of irrelevance.

8.2 Related dependently typed systems

Most of the following systems have already been discussed in Section 3.2.

8.2.1 Agda

Agda [Nor07] is a dependently typed programming language with full dependent pattern matching and irrelevance.

As described in Section 3.2.1.3, Agda's irrelevance is an *extrinsic* form of erasure but due to interaction with other language features, especially typed equality and η-conversion for records, it is not useful for erasure of indices (explained in detail in Section 2.1.7.2).

Zombie shows [Sjö15] that with an *untyped* equality, we lose η-conversion but irrelevance is usable in the way that we need for erasure of indices, and it will accept (the equivalent of) the problematic program shown in Section 2.1.7.2.

Irrelevance is fully explicit in the source code and there is no irrelevance inference.

8.2.2 Coq

Coq is a proof assistant and a programing language with dependent types.

Its pattern matching facility are case trees composed of **match** expressions. Since case trees do not automatically benefit from "rewriting in the context" provided by pattern clauses, and Coq's case trees do not have *smart case* [Sjö15], dependent pattern matching is more complicated and requires tricks like the convoy pattern [Chl13].

For example, let us take the following definition of function f.

```
Definition f✗ (P : unit → Type) (x : unit) (px : P x) : P tt :=
  match x with
  | tt ⇒ px — ERROR: "px" has type "P x" while it is expected to have type "P tt".
  end.
```

This definition is rejected because the value px has the type $P\ x$ instead of P tt, even though we are in the branch where x = tt so we know that $P\ x = P$ tt. We could also describe this problem as not rewriting in the context after a pattern match.

The solution in Coq is using the *convoy pattern* [Chl13], which yields a definition of f as follows.

```
Definition f (P : unit → Type) (x : unit) (px : P x) : P tt :=
  (match x with
  | tt ⇒ (fun px ⇒ px)
  end) px.
```

This pattern exploits the fact that even though Coq does not rewrite in the context, it *does* rewrite in the *goal*. We therefore change the **match** expression to return a function of the type $(P\ x \to P\ \mathsf{tt})$, which specialises to the desired $(P\ \mathsf{tt} \to P\ \mathsf{tt})$ in the branch $x = \mathsf{tt}$. We then apply this function to px.

For erasure, Coq uses a separate (impredicative) universe of types, Prop. Each inductive type (family) is defined either in Prop if its values should always be erased, or in Set (or Type) if its values should never be erased. Elimination of targets in Prop is not allowed for non-Prop motives, except for empty or singleton targets, where:

- A singleton definition has only one constructor and all the arguments of this constructor have type Prop. [The04]
- An empty definition has no constructors, in that case also, elimination on any sort is allowed. [The04]

This ensures that values in Prop can always be erased without losing information that might be necessary for execution of programs.

Determining erasure behaviour at the point of the definition of the inductive type family for all its values is called *intrinsic view of erasure* by Mishra-Linger [ML08].

The practical consequence of the intrinsic view of erasure is that the example programs in Section 3.1.2 are not satisfactorily erased, as explained in Section 3.2.1.2.

Finally, the decision what is erased is done manually by the programmer, by defining types in either Prop or Type and there is no erasure inference.

8.2.3 Zombie

Zombie [Sjö15] is a dependently typed language belonging to the Trellys project [CSW14].

The Trellys project is a search for a dependently typed language that features general recursion for convenient programming but also a sound logical fragment for trustworthy proofs at the same time.

The overarching theme is classification of functions as *logical* (terminating and consistent) or *programmatic* (general recursion allowed). The logical fragment, the proofs, is allowed to talk about the programmatic fragment, the programs, but it is not allowed to run it [Cas14]. This means that programmers are not forced to prove termination of *all* functions in a program, which makes sense since not all functions represent proofs.

The Trellys project has spawned several dependently typed languages, one of them being Zombie, which is a dependently typed language with erasure and inductive types, besides other features. Furthermore, its design with untyped equality avoids Agda's problems with irrelevance (Section 2.1.7.2), which makes Zombie the only current system that would be able to deal with the problems presented in Section 3.1.2.

Zombie does not have indexed type families. Instead, equality built into the language, together with *parameterised* inductive types, allows encoding indexed families in the Henry Ford style (see Section 7.2.2.3). $\mathsf{TT}_\star$, on the other hand, uses indexed type families and pattern clauses for dependent pattern matching.

Another difference to $\mathsf{TT}_\star$ is that Zombie makes no distinction between erasability and irrelevance and a value can be either both or none. Section 7.6 argues that, besides limited expressivity, equating erasure and irrelevance makes erasure inference more difficult. Zombie does not have erasure inference, and is therefore unaffected.

$\mathsf{TT}_\star$ has erasure inference, which removes the need for erasure annotations inserted by programmers. Zombie expects explicit irrelevance/erasure annotations in the surface language.

8.2.4 Dependent Haskell

Dependent types are coming to Haskell, too [Gun13; Eis16; Wei+17].

Eisenberg's design, like Zombie, makes erasure explicit in the surface language by letting the programmer to choose from a palette of 12 quantifiers, each offering a different combination of dependency, relevance, visibility and matchability [Eis16, Sec. 4.2.5, Fig. 4.1]. In the absence of an explicit choice, the default is the usual non-dependent, unerased, explicit binder ($\to$).

While $\mathsf{TT}_\star$ has erasure fully explicit in the $\mathsf{TT}_\star^{\mathsf{RE}}$ stage, too, it does not require any erasure annotations in the input $\mathsf{TT}_\star^\bullet$ stage, and it is expected that programmers will generally not erasure-annotate their programs explicitly, except for cases like FFI or compiler builtins.

To present type safety in the presence of non-termination, Dependent Haskell *executes* some proofs of type equality at runtime to ensure that type coercions are valid (or non-terminating). This is similar to the problem described in Section 9.2.1.8, where erasure of an unused and absurd argument of a function breaks strong normalisation, uncovering ill-typed reductions.

While in a total language, we must just ensure that the ill-typed reductions remain hidden inside a lambda at runtime, in Dependent Haskell, we must additionally check that the computation of type equality proofs terminates because the language does not guarantee it.

8.2.5 Cayenne

Cayenne [Aug99] is a Haskell-like language with dependent types aimed at practical programming, as shown by its pragmatic approach to various questions.

Cayenne has inductive types and type-annotated case expressions for elimination (see Section 7.2.2.4).

Furthermore, Cayenne erases exactly the types, which are all values with the type $\#_n$ for some universe level n. Non-types, including proofs and indices, therefore remain in the program at runtime.

8.3 Erasure and flow analysis

8.3.1 Useless variable elimination

The terminology around dead code elimination is not consistently established. Shivers [Shi91] defines *useless variables* as those whose value contributes nothing to the final outcome of the computation. Muchnick [Muc97] defines *unreachable code* as code that is never executed and *dead code* as code that is executed but its result is unused. Wand and Siveroni [WS99] define *dead variables* as variables that are not needed after a given point in the program. *Useless variables* are dead immediately after their initial binding. Berardi et al. [Ber+00] define *dead code* as code that is never executed and *useless code* as code that can be erased without affecting the final output of a program.

I will avoid using the ambiguous expression "dead code", in favour of "unreachable code" or "useless code".

Since uselessness is defined using liveness, live-variable analysis [Aho+06] is therefore related to useless variable analysis/elimination.

Shivers' UVE In his dissertation, Shivers sketches a useless variable elimination (UVE) algorithm that works in the same way as the erasure inference algorithm in Chapter 4 of this dissertation: find variables that *might* be useful, mark the rest as useless.

Chapter 4 elaborates the algorithm in detail, extending it with features necessary for practical functional programming, such as inductive type families, type classes, discussing efficient constraint solving.

Shivers does not explicitly mention (k-) CFA in the section about UVE but running CFA prior to erasure analysis would give a more precise estimate of usage patterns of functional variables. Currently, Chapter 4 does not use CFA and it uses the most conservative approximation, assuming that functional variables use *all* their arguments.

UVE with constraints Wand and Siveroni [WS99] describe a constraint-based UVE approach that builds on Shivers' dissertation and also uses 0-CFA. They observe that UVE does not preserve termination and effects (see also Section 9.2.1.8) and they propose marking all arguments that are not obviously free of effects as useful. This UVE approach does not discuss inductive types.

8.3.2 Other analyses

8.3.3 Control flow analysis

Shivers introduces CFA in his dissertation [Shi91]. This analysis can answer the question "Which lambdas/functions could possibly be referred to by this functional variable?" Further development of CFA is reviewed by Midtgaard [Mid12].

Such knowledge would make erasure inference in Chapter 4 more precise, since at the moment, it assumes that functional variables use *all* their arguments. With CFA, we could find arguments that are unused in all candidate functions and erase them.

Chapter 5 introduces $\mathsf{TT}_\star$, a calculus with erasure built into the type system, where CFA would not help with erasure inference. However, when extending $\mathsf{TT}_\star$ with erasure polymorphism of type families, CFA might help with issues arising from types being too imprecise (Section 7.3.2.5).

Strictness analysis Strictness analysis [Myc80; PJ87] looks for arguments that are certainly *used*, as opposed to being certainly *unused*. Traditionally, strictness analysis uses abstract interpretation, rather than constraints, and can analyse higher-order functions (unlike the approach presented in Chapter 4, without extensions). It is possible that strictness analysis could be adapted to perform erasure analysis, although it is unknown whether that would be easier than adding higher-order capabilities to the simple erasure analysis (Section 4.5.5).

Program slicing In imperative programming, program slicing is an analysis that, given a subset of a program's behaviour, finds a minimal subprogram that preserves the behaviour [Wei81]. In pure functional programs, erasure inference (and subsequent erasure) can therefore be viewed as the backward slice of the result of the main function.

8.4 Related calculi

A good overview of pre-2008 type-based erasure can be found in Mishra-Linger's dissertation [ML08, Chap. 7] and in the position paper by Berardi et al. [Ber+00].

One line of work stems from Martin-Löf's type theory with typed equality [MLS84], through Pfenning's irrelevance [Pfe01], resulting in irrelevance in Agda [AS12; Agd14], QTT [Atk18] and ParamDTT [NVD17].

Another line of work, based on pure type systems with untyped equality, includes Luo's UTT [Luo94], Miquel's Implicit Calculus of Constructions [Miq01], TT [MM04; Bra13], Barras and Bernardo's ICC* [BB08], Mishra-Linger's EPTS [ML08], CCCC [BM13], Zombie [Sjö15] – and $\mathsf{TT}_\star$.

The influence of typed equality is discussed in Section 2.1.7.2. In short, typed equality enables η-conversion and makes interpretation easier [AVW17] but also seems to restrict the ways that programs can depend on irrelevant values.

8.4.1 Linear types

8.4.1.1 Plenty O' Nuttin'

McBride [McB16] shows how to combine linearity with dependent types. In order to be able to derive useful type judgements after context splits, McBride introduces quantification (as in "how many?") in typing environments and judgements.

The typing rule for lambdas illustrates this approach.

$$\frac{\Gamma, \rho\pi\ x : S \vdash \rho\ T \ni t}{\Gamma \vdash \rho\ (\pi\ x : S) \to T \ni \lambda x.\ t}\ \textsc{App}$$

The App rule reads: "If in the environment Γ, extended with a supply of $\rho\pi$ copies of $x : S$, we can construct ρ copies of t with type T, then in Γ, we can construct ρ copies of $\lambda x.\ t$" with type $(\pi\ x : S) \to T$. The type $(\pi\ x : S) \to T$ says that the function requires π copies of $x : S$ to construct T.

This illustrates that typing environments keep track of *how many* "copies" each binding can supply – whatever the words "how many", "copy" and "supply" mean in the given context. The variables ρ or π stand for quantities, which form a *rig* (ring without negation).

McBride notes that the trivial rig $\{0\}$ yields the traditional type theory, and then gives a motivating example of a none-one-tons rig with three elements: 0, 1, and ω.

$\rho + \pi$	0	1	ω
0	0	1	ω
1	1	ω	ω
ω	ω	ω	ω

$\rho\pi$	0	1	ω
0	0	0	0
1	0	1	ω
ω	0	ω	ω

Table 8.1: The none-one-tons rig

Pi bindings annotated with these quantities generalise the familiar parametric, linear, and ordinary Pi binders.

Traditional notation	McBride's notation
$\forall x : S.\ T$	$(0\ x : S) \to T$
$(x : S) \multimap T$	$(1\ x : S) \to T$
$(x : S) \to T$	$(\omega\ x : S) \to T$

Keeping track of quantities lets us split any context into two without having variables disappear – it is only their quantities that split, not their presence. This also solves any reordering issues.

The typing rule for application illustrates this.

$$\frac{\Delta_0 \vdash \rho\ f \in (\pi\ x : S) \to T \qquad \Delta_1 \vdash \rho\pi\ S \ni s}{\Delta_0 + \Delta_1 \vdash \rho\ (f\ s) \in T[x \mapsto s : S]}\ \textsc{App}$$

$\Delta_0 + \Delta_1$ is pointwise sum of quantities in environments Δ_0 and Δ_1 and it is defined only for environments of the same "shape". Both environments Δ_0 and Δ_1 bind the same set of names as $\Delta_0 + \Delta_1$, but their quantities may be smaller.

The calculus is presented in a bidirectional way, which uses typing judgements with $\in$ and $\ni$. McBride proves several metatheoretic results, including subject reduction. Erasure produces run-time programs in the untyped lambda calculus, and McBride proves that erased programs respect resource quantities and "do not go wrong" (by proving that a post-erasure reduction corresponds to some number of pre-erasure reductions).

Extrinsic notion of quantity $\mathsf{TT}_\star$, like EPTS [ML08], builds on the extrinsic view of erasure: erasability of an expression is not a property of its type but it depends on the context in which the expression occurs.

McBride's quantification is similar. There is no notion of a "linear type" – instead, variables are *bound* linearly. This allows for more flexibility, just like with erasure.

Relative notion of quantity Again, like erasure in $\mathsf{TT}_\star$ and EPTS, McBride's quantification is relative to the context. If a function requires π copies of its argument to construct its result and we need to compute ρ copies of its result, then we need to supply $\rho\pi$ copies of the argument. This allows us to apply a runtime function to an erased value, as long as we do it in an erased context.

This is in contrast to absolute modalities like Prop in Coq, where a function like plus : nat → nat → nat, where nat lies in Type, will never accept a value from the Prop universe as an input.

Pi introduction and elimination McBride's system quantifies the lambda rule and the substitution rule in the spirit of the following unidirectional rules:

$$\frac{\Gamma, \rho\pi n : \sigma \vdash \rho T : \tau \qquad \ldots}{\Gamma \vdash \rho(\lambda \pi n : \sigma.\ T) : (\pi n : \sigma) \to \tau}\ \textsc{Lam} \qquad \frac{\Gamma, \pi n : \sigma \vdash \rho T : \tau \qquad \Gamma' \vdash \pi N : \sigma}{\Gamma + \Gamma' \vdash \rho T[n \mapsto N] : \tau[n \mapsto N]}\ \textsc{Subst}$$

Above, Lam places n into the environment with quantity $\rho\pi$, while the substitution lemma requires N at the same quantity as n – which is $\rho\pi$.

On the other hand, $\mathsf{TT}_\star$ uses the following lambda and substitution rule.

$$\frac{\Gamma, n :_s \sigma \vdash T :_r \tau \qquad \ldots}{\Gamma \vdash \lambda n :_s \sigma.\ T :_r (n :_s \sigma) \to \tau}\ \textsc{Lam} \qquad \frac{\Gamma, n :_s \sigma, \Gamma' \vdash T :_r \tau \qquad \Gamma \vdash N :_{r \wedge s} \sigma}{\Gamma, \Gamma'[n \mapsto N] \vdash T[n \mapsto N] :_r \tau[n \mapsto N]}\ \textsc{Subst}$$

In these rules, Lam places n into the environment with retention s, but the substitution lemma requires N with retention $r \wedge s$.

In both approaches, rules App, Lam and Subst therefore play together well to preserve types of terms under substitution, and thus under reduction, too.

Rig vs. lattice McBride's quantities form a rig. Erasure levels in this dissertation form a lattice.

Rig multiplication $\rho\pi$ in the none-tons rig, which omits the "one" quantity, coincides with $\wedge$ in the lattice $\mathsf{E} < \mathsf{R}$ if 0 corresponds to E and ω corresponds to R. Since the rig addition $+$ is used only in splitting contexts, and $\mathsf{TT}_\star$ does not split contexts, I did not define a corresponding operation, but with $\{0, \omega\}$ it would correspond to the operation $\vee$. Indeed, we can observe that this way, every (complete) distributive lattice can be interpreted as a rig.

Rig multiplication vs. lattice meet

Linearity If we ignore the quantity 1, which can split only unevenly, there is almost no difference between context copying and context splitting – which yields the non-context-splitting App rule of $\mathsf{TT}_\star$.

The only exception is that context-splitting with a rig allows splitting ω copies of a binding into ω in one context and 0 in the other context. This is not allowed in $\mathsf{TT}_\star$.

The quantity 1 can therefore *not* be modelled simply as an erasure level in $\mathsf{TT}_\star$ without adding context-splitting, too.

Weakening The (very limited) "subtyping" of $\mathsf{TT}_\star$ (Lemma 5.11) corresponds to McBride's rule Weak with $0 \leq \omega$, where McBride's ordering $\rho \leq \pi$ expresses that it's acceptable to use a π-bound variable only with quantity ρ.

Pattern matching McBride's calculus does not have pattern matching, **let** bindings, or similar syntactic elements; it deals only with (and proves metatheory only for) variables, lambdas and applications. Inductive type families are supplied externally using constructors and eliminators.

Metatheory McBride proves several useful metatheoretic results that I have not proven for $\mathsf{TT}_\star$, most importantly step simulation (Conjecture 5.2). It would be useful to adapt proof techniques from this paper for $\mathsf{TT}_\star$.

Demolition McBride uses the word *demolition* for the variant of elimination [McB02] where the target is quantified linearly. This allows the code to destructively manipulate the original data structure – to deconstruct it into pieces, possibly reusing these pieces to construct the output, since the type system statically ensures that the input is not needed anywhere else.

Summary On the intersection of both calculi (no pattern matching, no let bindings, no linear types, etc.), barring different presentation and cosmetic differences, $\mathsf{TT}_\star$ and McBride's system with weakening $0 \leq \omega$ are essentially equivalent.

8.4.1.2 Worlds

The above paper is clearly related to McBride's previous unpublished work (some together with Gundry) on *worlds* and phases [GM13; McB14a; McB14b], which also use "world" annotations on binders and rig-style multiplication, although not in a linear-typing context.

8.4.1.3 Quantitative Type Theory

McBride's work on linear types has been reformulated and extended as Quantitative Type Theory (QTT) by Atkey [Atk18], who observes that recording extra information with bound variables has been recognised as useful also elsewhere in the literature [POM14; Bru+14; GS14].

QTT features separate syntactic categories for terms and types; the universe of small types, Set; the Bool type with its values and eliminator. QTT records quantities for each bound variable, but also argument and return types and quantities for function applications. Atkey then establishes the metatheory of QTT, and gives a sound model of QTT.

In QTT, Atkey corrects a problem in McBride's system, where the typing rules allow η-expansion in ω-contexts to coerce a linear function into a non-linear one. The eta-expanded form then cannot be substituted into certain terms for a variable of the same type without breaking type-correctness of the resulting expression. To remedy this, QTT restricts term typing judgements to quantity 0 or 1; one cannot type a term at quantity ω.

Weakening QTT does not feature the rule Weak, as defined by McBride [McB16, Sec. 12], nor does it discuss any ordering on quantities at all. Weakening in QTT therefore allows us to add only 0-quantified names into the context without breaking judgement derivations. This means that $(\lambda x :_\omega \tau.\ x)$ is ill-typed in QTT since we cannot type terms at quantity ω, nor can we weaken the quantity of x inside the lambda.

For practical programming, QTT will likely need to be extended with the McBride-style weakening to allow at least using ω-quantified variables in 0-contexts, like $\mathsf{TT}_\star$ does.

8.4.2 Parametric Quantifiers for Dependent Type Theory

Vezzosi, Nuyts and DeVriese [NVD17] also extend Martin-Löf's Type Theory with parametric binders, in order to recover parametricity provided by System F, lost by the unification of $\forall$ and Π in type theory.

The resulting type theory, ParamDTT, has quantifiers (and lambdas) with three possible modalities: pointwise (¶), parametric (#), or continuous (id). The quantifiers $\forall$ and $\exists$ are then expressed as parametric (#) variants of Π and Σ, while the usual Pi and Sigma binders correspond to their continuous (id) variants.

ParamDTT defines relational structure on types in two levels: bridges and paths. Paths represent (possibly heterogeneous) equality and bridges represent (possibly heterogeneous) relatedness. All functions must respect paths, and the difference between the three binder modalities lies in how they respect bridges:

- Parametric functions map bridged elements to path-connected elements;
- continuous functions map bridged elements to bridged elements;
- pointwise functions have none of the above constraints.

Furthermore, free theorems can be derived in ParamDTT *internally*.

As mentioned in Section 2.1.7, ParamDTT allows the definition of a *parametric* identity function $f : \mathbb{N}^{\#} \to \mathbb{N}$. This is possible because the natural numbers, as defined in ParamDTT, do not have any (non-identity) bridges, and this trivial relational structure is respected by any function trivially.

Since the argument of f clearly cannot be erased, ParamDTT demonstrates that irrelevance is different from parametricity. My dissertation argues that it is useful to consider also *erasure* as distinct from both irrelevance and parametricity.

8.4.3 Shape irrelevance

Abel, Vezzosi and Winterhalter [AVW17] discuss termination checking with *sized types* in a type theory with typed equality. A type is sized by indexing it with a size, like in the following example introducing sized natural numbers.

```
data Size : Type where
  SZ : Size
  SS : Size → Size

data ℕ : (i : Size) → Type where
  Z : {i : Size} → ℕ i
  S : {i : Size} → ℕ i → ℕ (SS i)
```

The problem with sizes is that the explicit presence of sizes in terms interferes with definitional equality. For example, two differently sized versions of zero, (Z SZ) and (Z (SS SZ)), are not definitionally equal, while we would like them to be.

In a calculus with untyped equality, such as ICC* or Zombie, the solution is easy – make the size argument to the *type* constructor *relevant*, while making the size arguments to the *data* constructors *irrelevant*, as discussed in Section 2.1.7.1.

```
data ℕ : (i : Size) → Type where
  Z : .{i : Size} → ℕ i
  S : .{i : Size} → ℕ i → ℕ (SS i)
```

Given the above definition of $\mathbb{N} : \mathsf{Size} \to \mathsf{Type}$, everything works as expected, $\mathbb{N}\ \mathsf{SZ} \not\approx \mathbb{N}\ (\mathsf{SS}\ \mathsf{SZ})$ but $\mathsf{Z}\ \mathsf{SZ} \approx \mathsf{Z}\ (\mathsf{SS}\ \mathsf{SZ})$ – not because we make a distinction between

terms and types or because we are checking equality as terms vs. types, but simply because the compared terms contain irrelevant *applications* (not indicated in the surface syntax here), which make their right-hand sides ignored in equality.

However, in a type theory with typed equality, the above definition is problematic in the presence of η conversion and large elimination, and is therefore rejected by Agda (Section 2.1.7.2).

The paper, together with its implementation in Agda, therefore presents an additional modality for Pi, *shape irrelevance*, which comes with the corresponding shape-irrelevant application, and for which large elimination is disallowed. This modality is denoted by *two* dots in the surface syntax and it can be used to mark type constructor arguments that may receive irrelevant values but are not ignored in equality. This yields the following definition.

$$
\begin{aligned}
&\textbf{data}\ \mathbb{N} : ..(i : \mathsf{Size}) \to \mathsf{Type}\ \textbf{where}\\
&\quad \mathsf{Z}\ :\ .\{i : \mathsf{Size}\} \to \mathbb{N}\ i\\
&\quad \mathsf{S}\ :\ .\{i : \mathsf{Size}\} \to \mathbb{N}\ i \to \mathbb{N}\ (\mathsf{SS}\ i)
\end{aligned}
$$

Unlike in untyped-equality calculi, resurrection is restricted to only arguments of shape-irrelevant applications.

8.4.4 Type Theory in Color

Bernardy and Moulin [BM13] present a calculus with colours and taints (sets of colours). Each binding is annotated with a modality: a pair (θ, ι) of taints, where θ gives the taint of the variable, while ι determines the obliviousness of the binding – the colours that are incompatible with substitution for that variable.

Expression are typechecked as tainted with a certain taint in the given environment and there is an erasure operation $\lfloor\cdot\rfloor_i$ that, given a colour i, erases all i-tainted portions of the given object (environment or term).

The above is illustrated by the following rule, which defines what it means for an expression to be θ-tainted and ι-oblivious.

$$\Gamma \vdash A :_{\theta,\iota} B \quad := \quad \lfloor\Gamma\rfloor_\iota \vdash A :_\theta B$$

Even if we remove all bindings that share at least one colour with ι from Γ, then A is still well-typed with taint θ.

The authors do not discuss inductive families, except for a mention that the work extends straightforwardly to inductive definitions.

Colouring and tainting goes far beyond a simple distinction between erasability and non-erasability, although erasure could be modelled by using a single colour R: every expression and variable that is tainted with R must survive until runtime.

It would be interesting to see how "taint inference" would work for an unlimited number of colours in CCCC. As the authors note, one cannot go wrong by using *more* colours and it would be interesting to see whether there exists a "finest colouring"

for every program that would use the maximum number of distinct colours while keeping the program taint-consistent. We could then conflate (or ignore) colours to obtain other taint-consistent programs.

8.4.5 Other

8.4.5.1 Security types

In imperative languages, security type systems [Den76; VS97; SM03] formalise the idea that expressions at a lower level of security should not be able to refer to values at a higher level of security. This is similar to our principle that runtime values should not refer to erased values.

Chapter 9

Results

This chapter demonstrates that the promises given in the introductory chapter have been fulfilled by the material presented in the rest of the dissertation, as summarised in Section 9.3.

9.1 Benchmarks

This section presents experimental data illustrating how erasure influences runtimes of the compiler and the compiled programs. Here, I discuss only the erasure method described from Chapter 5 onwards; Chapter 4 has separate benchmarks in Section 4.6.1.2.

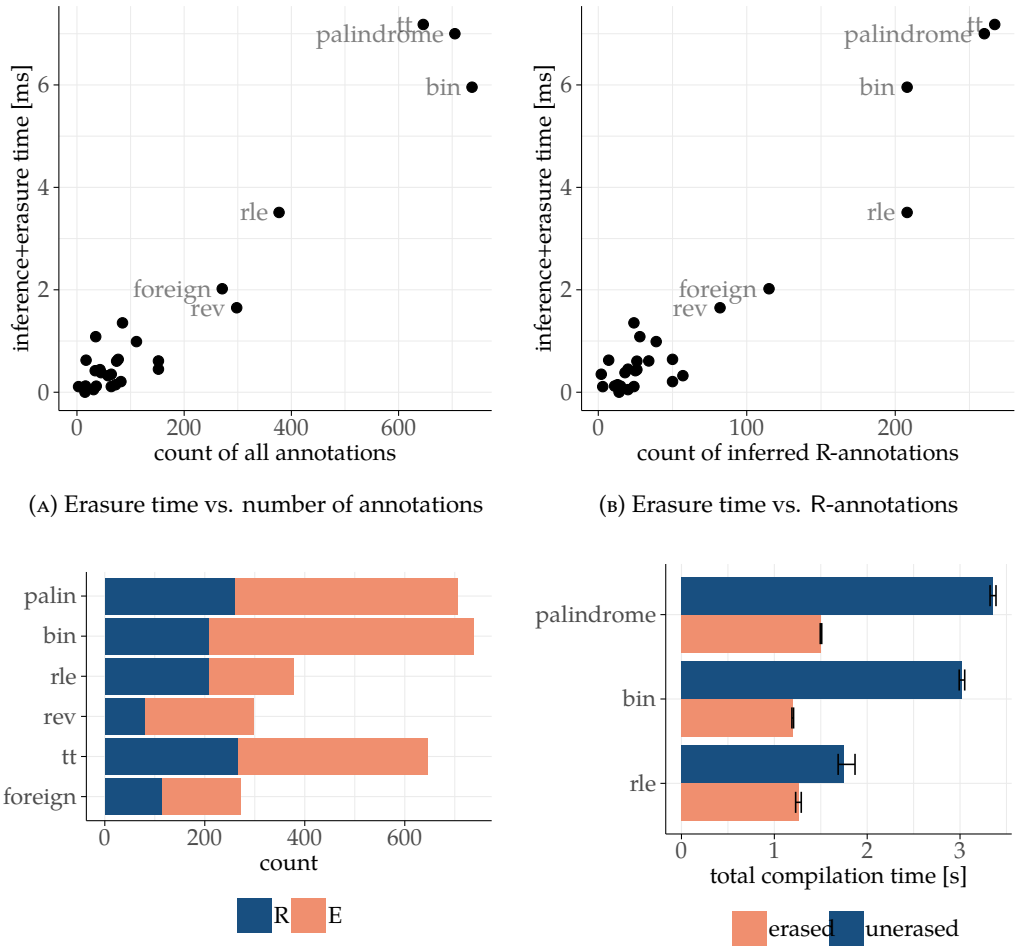
9.1.1 $\mathsf{TT}_\star$ compiler pipeline

My implementation of $\mathsf{TT}_\star$ uses the following steps/stages.

1. The input of the compiler is a hand-written program in the core calculus $\mathsf{TT}^{\bullet}_{\star}$.
2. The compiler first parses the program and then its subsequent stages correspond to Figure 5.5.

   Erasure polymorphism is implemented but disabled and ignored in these benchmarks.
3. The compiler outputs Scheme code, which is a more or less direct translation of the resulting program in its untyped $\mathsf{TT}^{\square}_{\star}$ form; it is curried and higher-order.
4. Pattern matching is translated using the matchable extension of Chicken Scheme (also present in Racket).

   My implementation can compile pattern clauses to case trees but this feature is ignored in these benchmarks[1].
5. The Scheme code is interpreted or further compiled to native code with the highest optimisation level, using Chicken Scheme [CHI20] in both cases.



(A) Erasure time vs. number of annotations

(B) Erasure time vs. R-annotations

(C) Inferred annotations

(D) Total compilation time including erasure
Error bars show minimum and maximum.

FIGURE 9.1: Erasure at compile time
Error bars (omitted) would be generally smaller than the point markers.

9.1.2 Compile time

Figure 9.1 illustrates how erasure influences the compiler using the following four plots.

9.1.2.1 Time taken by erasure

Figure 9.1a talks about the run time of the $\mathsf{TT}_\star$ stage, which includes (sub-)stages from parsing $\mathsf{TT}^\bullet_\star$ up to and including generating Scheme. Specifically, it does not include the Chicken Scheme compilation stage, nor does it include the execution of the resulting program.

The visualised quantity in Figure 9.1a, called "erasure time", is the difference between the runtime of the $\mathsf{TT}_\star$ stage with and without erasure inference enabled. More precisely, "without erasure inference", all erasure annotations are set to R instead of running erasure inference and the processing proceeds normally afterwards.

The horizontal axis represents the count of all erasure annotations. This number consists mostly of the number of variable binders in the program and the number of function applications in the program.

The benchmarked programs are small test programs from the test suite of my implementation of $\mathsf{TT}_\star$.

foreign is a FFI test that implements decidable equality for lists

rev implements the reverse view for lists (Section 2.2.8.2)

rle implements RLE compression and decompression

bin implements the binary adder

tt implements an evaluator for a scope-indexed lambda calculus

palindrome implements the palindrome check, using the reverse view as one of its components

The rest are various tiny programs that typically test a single language feature each.

Figure 9.1c shows how all annotations in the individual programs, as computed by erasure inference, were divided between retained and erased.

Here, it is interesting that despite the fact that of all programs, bin contains the largest number of erasure annotations in total, the subset of them inferred to be retained is exactly the same as that of the relatively small program rle.

The number of all annotations and the number of R-annotations seem to be equally good predictors of time spent by inference and erasure in this experiment. With a linear model, both achieve $r^2 \approx 0.91$.

[1] It does not seem to make significant difference in the performance of native code.

9.1.2.2 Total compilation time

Finally, Figure 9.1d shows total compilation time, including the $\mathsf{TT}_\star$ stage *and* the Chicken Scheme compiler (at the highest optimisation level).

This demonstrates that even if, as in the case of palindrome, erasure makes the $\mathsf{TT}_\star$ stage take 7 milliseconds longer, the total compilation time, which includes the subsequent stages, is shortened by more than 1500 miliseconds.

These speedups apply to my implementation of $\mathsf{TT}_\star$ with the Chicken Scheme backend. Figure 4.9 in Section 4.6.1.2 shows that in the case of Idris, erasure led to much less impressive savings in these example programs.

9.1.3 Execution time

Figure 9.2 shows how erasure affects the run times of our three example programs.

9.1.3.1 Palindrome

Input (Unary) natural number representing the input size.

The program internally builds a list of Booleans with alternating True and False.

For the benchmarks, the input sizes are always odd so that the resulting list is always a palindrome (and needs to be checked completely).

Operation The program checks whether the list is a palindrome.

Output The program prints True or False.

Figure 9.2a shows how the run time of the palindrome checker depends on the input size (length of the input list) – and erasure.

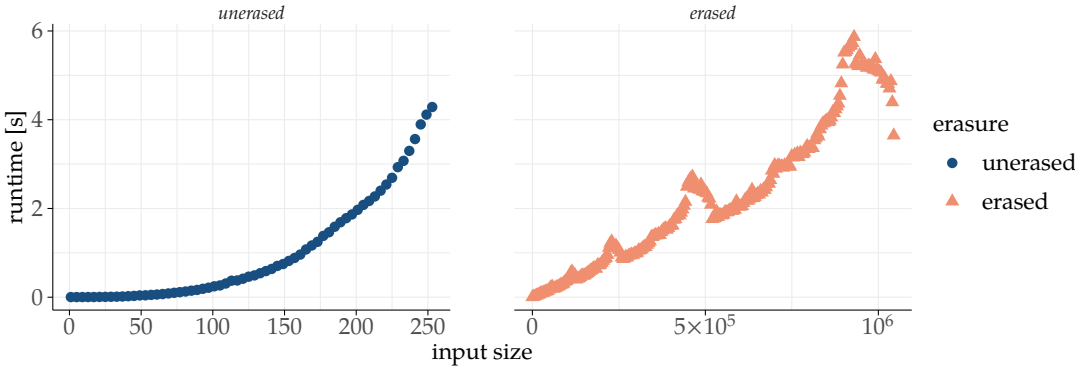
First of all, note that the two plots have very different horizontal scales – the unerased program is run for inputs having up to 250 elements, while the erased program is run for inputs with up to 1 million elements to obtain comparable runtimes.

The unerased program is obviously convex and non-linear. It is harder to judge the erased program, whose plot contains significant spikes. Since the spikes seem to appear log-regularly (at about powers of two times 1.25×10^5), and they appear in the same places with the same shape across different machines, I expect that they are caused by heap resizing or other Chicken RTS effect. I have not explored these effects further.

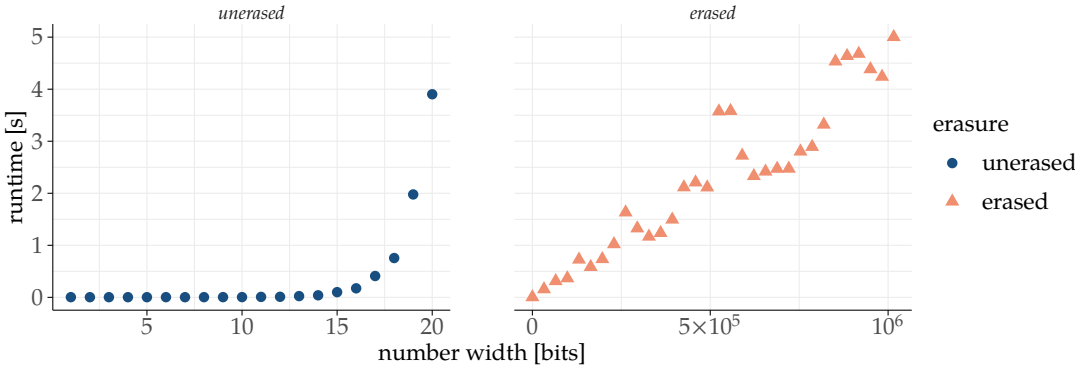
Section 9.1.4 gives more concrete estimates on the asymptotics of these programs.

9.1.3.2 Binary adder

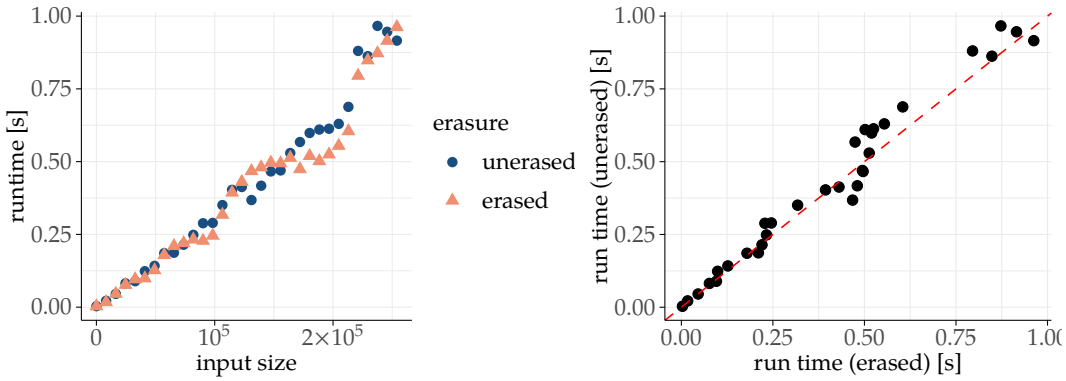
Input Unary number representing the input size. The program internally builds two binary number with the given size. The two binary numbers always consist of alternating ones and zeroes, and are complementary to each other. (Building these unary numbers is included in the runtime.)



(A) Palindrome checker



(B) Binary adder



(C) RLE

(D) RLE: closer look

FIGURE 9.2: Erasure at run time
Error bars (omitted) would be generally smaller than the point markers.

Operation The program adds the two binary numbers built in the input step.

Output Program prints the resulting bits of the sum. (More precisely, it prints the internal Scheme representation of the binary number, which is a right-nested list.)

Figure 9.2b shows that the runtime of the unerased program roughly doubles with every additional bit.

On the other hand, the compiled version of the binary adder looks approximately linear. It also seems to exhibit similar behaviour as the compiled version of the palindrome checker. Their runtimes are roughly the same on the same input sizes and both programs' run time also spikes around similar input sizes.

This is quite remarkable, given the very different asymptotic behaviour of the unerased versions of these two programs.

(The input size step was 8× larger in the case of the binary adder than the palindrome checker.)

9.1.3.3 RLE

Input Unary number representing the input size.

The program internally builds a Boolean list of the given size, with all elements being True.

Operation The program RLE-compresses and decompresses the input list, and then XORs all elements of the list together using a left fold.

Output The program prints the resulting value, True or False.

In this program, erasure does not lead to an *asymptotic* improvement, as seen in Figure 9.2c; it just removes some constant overhead. This allows us to use the same input sizes for both erased and unerased programs, unlike in the previous examples.

From the plot, it is not immediately obvious that the erased program is faster. This is likely because it is easy to visually mis-pair the markers (the correct pairing always pairs markers for the same input size – vertically).

For that reason, I also include Figure 9.2d, which plots the erased run time versus the unerased run time for each input size. The red dashed line indicates the graph of the identity function and in this plot, it is more visible that the unerased time for larger-sized inputs is usually larger than the erased time.

Finally, the paired t-test[2] rejects the null hypothesis that erasure had no effect on the run time of this program at $p \approx 2\%$.

[2]One might argue that the paired t-test gives more weight to higher input sizes (which cause greater absolute differences in run times), but we could counter-argue that this bias is in fact desirable. I have not attempted to correct for it.

9.1.4 Exponent estimation

Assuming that the run time is a polynomial function of input size, we can estimate the order of the polynomial as the slope of the regression line in the log-log plot of run time vs. input size. These plots are shown in Figure 9.3.

This approach is justified by the following derivation, where the run time $T(n)$ depends on input size n.

$$\begin{aligned} T(n) &= \beta n^\alpha + o(n^\alpha) \\ \log T(n) &= \log\left[\beta n^\alpha \cdot \left(1 + \frac{o(n^\alpha)}{\beta n^\alpha}\right)\right] \\ \log T(n) &= \alpha \log n + \log\beta + \log\left(1 + \frac{o(n^\alpha)}{\beta n^\alpha}\right) \end{aligned}$$

As the input size n grows, the residual term $\log\left(1 + o(n^\alpha)/\beta n^\alpha\right)$ vanishes, yielding a linear curve with slope α in the log-log plot.

One needs to be careful with experimental big-oh estimation in general. For example, finer distinctions, such as $\log n$ factors, are hard to determine experimentally and a finite set of samples cannot *prove* anything about the asymptotic behaviour of a program, anyway – we could always simply say that the claimed asymptotic behaviour sets in only for higher input sizes than those present in the experiment [MPC97; SF00].

However, I will use this method to estimate only the highest exponent of a polynomial, to informally illustrate that the experimental complexity follows the predicted values.

9.1.4.1 Palindrome

Figure 9.3a shows linear regression in the log-log plot for the palindrome decider.

To reduce the effect of smaller-order terms, I (arbitrarily) excluded all samples with input sizes smaller or equal to 40 elements from the exponent estimation. This affects only the unerased case and the cut-off limit is illustrated by a dashed vertical line in the plot. This corresponds to the High-End Power Rule [MPC97].

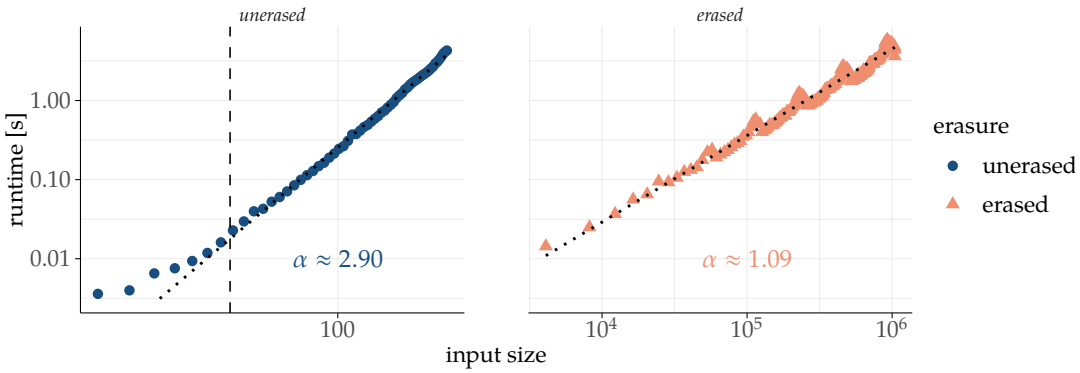
Like in Section 4.6.1.2, my implementation of the palindrome checker seems to be cubic. Again, this contradicts the expected complexity $O(n^2)$, unless it is relaxed to $\Omega(n^2)$.

After erasure, the program becomes approximately linear, as expected and desired.

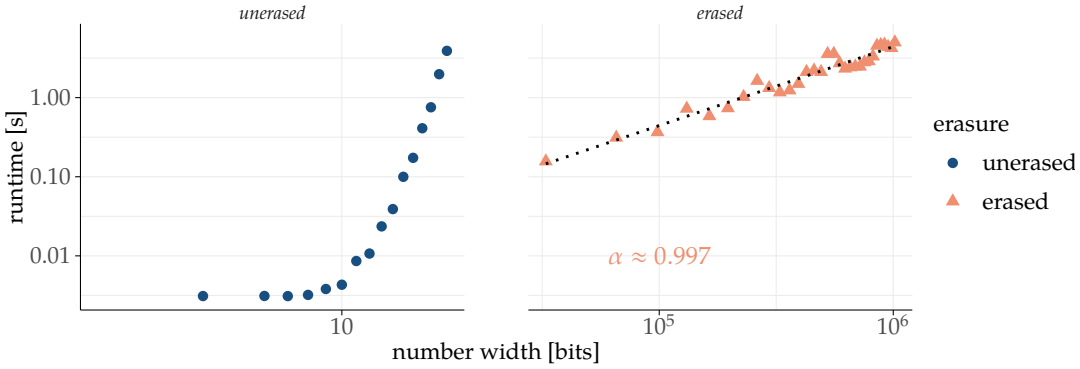
With logarithmical axes, the spikes in runtime become regularly spaced, which supports the hypothesis that this effect could be caused by a process similar to an exponentially growing heap.

9.1.4.2 Binary adder

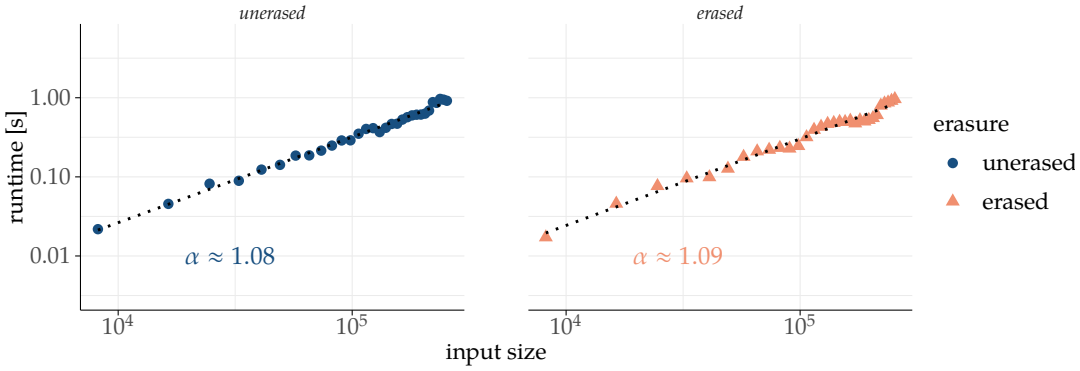
I do not perform regression in the unerased case since it does not satisfy the assumption of polynomiality.



(A) Palindrome checker



(B) Binary adder



(C) RLE

FIGURE 9.3: Log-log plots

Program	Expected complexity		Measured exponent	
	erased	unerased	erased	unerased
Palindrome checker	$O(n)$	$O(n^2)$	1.09	2.90
Binary adder	$O(n)$	$O(2^n)$	1.00	—
Run-length decoder	$O(n)$	$O(n)$	1.09	1.08

TABLE 9.4: Complexities of the example programs
Standard errors (not given) are on the order of 10^{-2}.

The erased program seems to run in linear time, as expected.

9.1.4.3 RLE

In the case of RLE, both erased and unerased programs have the same asymptotic complexity, which is supported by exponent estimation.

The fact that the erased program is slightly faster manifests as a slightly lower intercept (not visualised).

9.1.5 Summary

Like in Chapter 4, Section 4.6.1.2, my implementation of the palindrome checker is cubic when unerased and linear when erased, and all erased programs are linear, as expected (and desired). This is summarised in Table 9.4.

Therefore, the results obtained by exponent estimation indicate that also the erasure approach presented in Chapters 5 and 6 is successful in recovering the desired asymptotic complexity of all example programs.

9.2 Discussion

9.2.1 Remarks

9.2.1.1 Explicit annotations unnecessary

The approaches presented in this dissertation generally do not require explicit erasure annotations to work correctly.

An exception are only cases like foreign functions, where the compiler cannot infer the erasure pattern, or the implementation of the IO monad, where we want to prevent erasure inference from spotting that RealWorld is never used. In both cases, annotations are kept to absolute minimum (Section 7.5).

With the erasure method described in Chapter 4, which is implemented in Idris, the set of programs accepted by the compiler becomes strictly bigger: the programs that passed the compiler before adding erasure are all accepted after adding erasure, plus the compiler allows annotated programs. This means that a programmer could take the same program, compile it unmodified with a version of Idris that supports erasure, and obtain a program that is potentially (asymptotically) faster.

While the erasure method described from Chapter 5 onwards does not have a "before", non-erasure implementation, it still does not require any erasure annotations.

9.2.1.2 Forced patterns and erasure annotations

One might argue that forced patterns and forced constructors are a form of erasure annotations – we even (ab)use erasure annotations (Section 4.5.1.1) to choose forced patterns in Idris!

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.

9.2.1.3 Not running the programs that we write

Dependently typed systems have put a lot of effort into type inference, program inference, proof search, and similar kinds of automation. If we add erasure, we may get a situation where the program that we run is entirely different from the program that we write – if the programmer-provided code is used only to infer other code, and then erased.

This is already almost happening, and we don't even need advanced erasure. Currently, Idris will successfully infer both right-hand sides of the function zipWith, and Agda's proof search will even do the case splitting automatically.

$$
\begin{aligned}
&\mathsf{zipWith} : (a \to b \to c) \to \mathsf{Vect}\ n\ a \to \mathsf{Vect}\ n\ b \to \mathsf{Vect}\ n\ c \\
&\mathsf{zipWith}\ \ f\ \ \ \mathsf{Nil}\ \ \ \ \ \mathsf{Nil}\ \ \ = \mathsf{Nil} \\
&\mathsf{zipWith}\ \ f\ \ (x :: xs)\ \ (y :: ys)\ \ = f\ x\ y :: \mathsf{zipWith}\ xs\ ys
\end{aligned}
$$

We can therefore just write the type signature of zipWith, have its body inferred and compiled to machine code, while the type signature gets erased before code generation.

Note that this would not work for zipWith operating on lists because the type signature is not precise enough – there would be nothing stopping proof search from simply filling in empty lists on the RHS in both clauses. Therefore, (useful) proof search is made possible by the precision of dependent types.

With more precise types, smarter proof search and smarter erasure, we will probably encounter more interesting examples of this behaviour.

9.2.1.4 The core language is $\mathsf{TT}^{\mathsf{RE}}_{\star}$

Chapter 5 says that $\mathsf{TT}_{\star}$ is a core language. In fact, the core language is the variant of $\mathsf{TT}_{\star}$ called $\mathsf{TT}^{\mathsf{RE}}_{\star}$. In $\mathsf{TT}^{\mathsf{RE}}_{\star}$, everything, including erasure, is fully explicit and determined. Furthermore, $\mathsf{TT}^{\mathsf{RE}}_{\star}$ is the target language of all extensions to erasure inference, including erasure polymorphism (Section 7). Finally, $\mathsf{TT}^{\mathsf{RE}}_{\star}$ is the calculus used for the last stage of the erasure pipeline before erasure itself and it is the calculus checked by the rules given in Section 5.5.

9.2.1.5 Pattern checking

In Idris, patterns are checked by simply interpreting them as terms and typechecking them as such. This would not work in $\mathsf{TT}_{\star}$, where we have (a restricted form of) erasure subtyping.

$$
\begin{array}{l}
\textbf{data}\ \mathsf{ErasedNat} : \mathsf{Type}\ \textbf{where} \\
\quad \mathsf{Erase} : (x :_{\mathsf{E}} \mathbb{N}) \to \mathsf{ErasedNat} \\
\\
f :_{\mathsf{R}} (x :_{\mathsf{R}} \mathbb{N}) \to \mathsf{ErasedNat} \\
(x :_{\mathsf{R}} \mathbb{N}) \\
\quad f\ \widehat{}_{\mathsf{R}}\ x = \mathsf{Erase}\ \widehat{}_{\mathsf{E}}\ x \quad \textit{— consistent} \\
\\
g :_{\mathsf{R}} \mathsf{ErasedNat} \to \mathbb{N} \\
(x :_{\mathsf{R}} \mathbb{N}) \\
\quad g\ \widehat{}_{\mathsf{R}}\ (\mathsf{Erase}\ \widehat{}_{\mathsf{E}}\ x) = x \quad \textit{— ERROR: inconsistent}
\end{array}
$$

In both functions, f and g, the pattern variable x is R-bound.

If $(\mathsf{Erase}\ \widehat{}_{\mathsf{E}}\ x)$ appears on the RHS of the clause, like in the function f, we want to allow it because it makes sense – we are passing a runtime value *into* an erased position.

However, on the LHS of a clause, in the pattern in g, applying the constructor Erase to the (runtime) pattern variable x means that we are attempting to *read* a runtime value *from* an erased position – and this should certainly be disallowed.

Therefore, the same "expression", $(\mathsf{Erase}\ \widehat{}_{\mathsf{E}}\ x)$, should be checked differently depending on whether it occurs in a pattern or in a term. This however makes sense – in patterns, the flow of information is reversed, flowing *into* pattern variables, while in terms, information flows *from* variables.

This is exactly the reason that Section 5.5.2.3 introduces a separate set of typing rules for patterns.

9.2.1.6 Mutual recursion

For simplicity, $\mathsf{TT}_{\star}$ does not feature mutual recursion in order to focus on the central ideas of erasure. Section 2.2.4 shows several ways of encoding mutual recursion in

$\mathsf{TT}_\star$ and since these work well with erasure, there is no reason to expect problems with real mutual recursion.

My implementation used to support mutual recursion using a form of let bindings that could bind multiple mutually recursive definitions at once. Erasure *inference* (not checking) had to be iterated for every mutual block until the set of constraints reached a fixed point. This was necessary because while checking earlier definitions in the block, the later definitions have not been assigned any constraints yet. Erasure *checking* does not change with mutual recursion.

With the approaches in Section 2.2.4, it is not necessary to iterate erasure inference because all mutually recursive functions are united in a single (non-mutually) recursive definition. Pretending that a mutually recursive collection of functions has been defined using one of the ways in Section 2.2.4 for the purposes of erasure inference, but compiling them as true mutually recursive functions, might be the most efficient strategy. I have not explored this further.

9.2.1.7 Case trees vs. pattern clauses

I believe that case trees are more practical for a core language than pattern clauses, for reasons given in Section 7.2.1.

$\mathsf{TT}_\star$ has pattern clauses only to remove the conversion from case trees to pattern to make proofs easier.

9.2.1.8 Token type target elimination

It is known [ML08; Let03; Let04; Let08] that unrestricted erasure of void targets and token type targets in elimination, which includes erasure of equality proofs, does not preserve strong normalisation and therefore should not be allowed.

Indeed, Mishra-Linger [ML08, p. 133] gives a very short example illustrating the problem. This example includes a function that takes a false assumption which gets erased, uncovering a diverging term.

```
data TyEq : Type → Type → Type where
  TyRefl : TyEq a a

coerce : TyEq a b → a → b
coerce ⌈TyRefl⌉ = λx. x

sym : TyEq a b → TyEq b a
sym ⌈TyRefl⌉ = TyRefl
```

$$
\begin{aligned}
&\mathsf{loopy} : (a : \mathsf{Type}) \to (b : \mathsf{Type}) \to \mathsf{TyEq}\ a\ (a \to b) \to b \\
&\mathsf{loopy}\ a\ b\ eq = \\
&\quad \textbf{let} \\
&\qquad w : (x :_{\mathsf{R}} a) \to b \\
&\qquad w\ x = (\mathsf{coerce}\ eq\ x)\ x \\
&\quad \textbf{in}\ w\ \big(\mathsf{coerce}\ (\mathsf{sym}\ eq)\ w\big)
\end{aligned}
$$

The erased version of the above program contains a definition of loopy that has no normal form.

$$\mathsf{loopy} = \textbf{let}\ w = \lambda x.\ x\ x\ \textbf{in}\ w\ w$$

This happens because the false assumption is not inspected anywhere and is therefore erased.

Explicit annotation In the example above, the function $w : (x :_{\mathsf{R}} a) \to b$ has been manually annotated so that x is not inferred as erased. This is necessary to preserve the inconsistency. Without this annotation, the inference algorithm (Chapter 6) determines x to be erasable, and loopy becomes an identity function after erasure.

This of course does not diminish the severity of the problem – adding (consistent) annotations should not make a well-behaved program "go wrong".

Erasability of problematic definitions One could argue that the whole point of false assumptions is to make certain functions *not callable*. We can promise to provide a Void eliminator because we know, assuming consistency of the calculus, that all code paths invoking it are unreachable code and thus we never have to generate code for it. In other words, all "problematic" functions should be recognised as unused and erasable, thus not present in the erased code at all.

This is however not true. To trigger the undesired behaviour, we do not have to *invoke* loopy. Since the runtime representation of the function $(\lambda x :_{\mathsf{E}} \tau.\ .M)$ is exactly the same as that of M, any *unapplied* occurrence of loopy – which is of course freely constructible even in consistent calculi – will be a diverging term.

Solutions Chapter 4 presents an erasure transformation that does not remove arguments of functions. Instead, in each erased application, it replaces the provided argument expression with the special symbol □. This method therefore does not suffer from this problem.

Letouzey observes that these "problematic" invocations are rather rare [Let08, 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 [Let03, Sec. 2.1]. Compared to the previous paragraph, this method has the advantage of removing almost all overhead of dummy arguments, and more readable generated code.

Other effects In non-pure languages, erasure may not preserve effects other than just non-termination. Various authors [FH01; WS99] propose a separate effect check, marking all possibly effectful expressions as non-erased.

9.2.1.9 Forced patterns and unification

Forced constructors correspond to single-branch case splits (Section 7.2.3.2). We can use suggestive syntax sugar for single-branch splits in Section 7.2.3.5, together with a special form of case trees that allow terms in patterns, to illustrate that single-branch splits – and by extension, forced patterns – also correspond to (solved) pattern unification constraints and Henry-Ford-style [McB00] equalities.

9.2.1.10 Direction of data flow

In terms, variables are data sources; in patterns, pattern variables are data sinks. Since usage of data sources is determined by the usage of data sinks, we need to care about the direction of data flow and this leads to separate sets of type checking rules for patterns and terms (Section 5.5.2.3).

9.2.1.11 Multiple erasure levels

Erasure presented in this dissertation has two levels: erased and unerased. However, it might be useful to add more erasure levels, as literature shows – McBride talks about *worlds* [McB14a; McB14b], quantities [McB16], and phases [GM13] (with Gundry), security type systems talk about security categories and access classes [Den76; VS97], type theory in colour [BM13] talks about taints, multi-stage programming [Tah04] talks about stages.

In all of the above, including erasure, each value, independently of its type, "comes from" a certain world, stage, phase, or colour, and we want to restrict which world, stage, phase, or colour it could "travel to" – and it seems to be useful to have several of them.

I have not explored adding more erasure levels but I took care to formulate erasure constraints and the erasure inference procedure in Chapter 6 in terms of a general complete lattice of erasure annotations, while working with just the lattice $\mathsf{E} < \mathsf{R}$ in this dissertation.

In the big picture, erasure inference calculates the "lower bound" on the erasure level of a value as the meet of the lower erasure bounds of its data dependencies and, after suitable modifications to the solver, I believe it would work with erasure lattices richer that $\mathsf{E} < \mathsf{R}$.

Horn clauses This would be a further departure from Mishra-Linger's system [ML08], which models constraints as Boolean formulas in CNF. This dissertation makes the transition from CNF ($\neg a \vee \neg b \vee c$) to Horn clauses ($a, b \longrightarrow c$). Further transition to

lattice-based constraints $\big((a \wedge b) \leq c\big)$ would naturally generalise Boolean constraints to richer lattice structures of erasure levels.

9.2.1.12 Erasure constraints as part of type signatures

As explained in Section 7.3.1.9, it is useful to regard erasure constraints as part of type signatures since they define the conditions under which a definition (and its type) is erasure-consistent. Furthermore:

- Sometimes, a set of constraints may be more efficiently expressed extensionally, as the set of its models / consistent erasure patterns of the definition (Section 7.3.1.9).
- In order to *use* (apply) a definition, we need the constraints only to relate the evars in the type of the definition. This means that the constraint set can be simplified/reduced to elide the internal nodes, in order to get better inference performance (Section 7.3.1.9).
- In order to *specialise* a polymorphic definition into its monomorphic copies, we need the whole set of constraints.
- For some functions, their constraint sets may be reduced to a concrete assignment of R and E to evars ahead of time (Section 7.4).

9.2.1.13 Implementation

Erasure constraints I found it useful to represent erasure constraints not as Horn clauses in the form $(G \longrightarrow r)$ but as Horn clauses aggregated by their LHS, in the form $G \longrightarrow R$. The constraint $G \longrightarrow R$ stands for the set of (elementary) constraints $\{G \longrightarrow r \mid r \in R\}$.

Constraint sets can then be represented as finite maps from sets of prerequisites to sets of consequences, which makes the set deduplicated, and it allows us to quickly search for the set of consequences associated with the empty set of prerequisites.

The implementation in Idris represents constraint sets as finite maps from sets of prerequisites to finite maps from sets of consequences to sets of reasons explaining where the constraints come from (Section 4.5.6).

"Everything is a definition" In my implementation, all name binders, be it lambda-bound names, let-bound pattern matching definitions, or pattern variables, are represented the same way – as a definition (Section 5.2.2). This is convenient for the implementation but also for presentation of the typing rules.

$\mathsf{TT}_\star$ and types $\mathsf{TT}_\star$ is a family of calculi $\mathsf{TT}^{\bullet}_\star$, $\mathsf{TT}^{\mathsf{evar}}_\star$, $\mathsf{TT}^{\mathsf{RE}}_\star$, and $\mathsf{TT}^{\square}_\star$, and it has been very helpful to keep them separate in the implementation, as well. In my Haskell implementation, the TT type is parameterised by the type of erasure annotations.

Using further parameters, such as the constraint set representation, which also changes along the compiler pipeline, was however too laborious and later reverted.

Native code generation My implementation can generate Scheme code in various ways.

Using the matchable extension of Scheme, the code can can be compiled to native binaries using Chicken Scheme, or interpreted by Chicken Scheme or Racket. This alternative does not require compiling pattern clauses and the generation of Scheme is very straightforward.

With a pattern compiler, my implementation can also generate standard Scheme from $\mathsf{TT}_\star$.

9.3 Summary

This dissertation supports the following thesis.

> *Erasure in practical dependently typed programming is useful and feasible.*

Erasure is useful *There are dependently typed programs that are elegant and idiomatic but inefficient without erasure. Furthermore, there are whole programming techniques, such as programming with dependent views, that require erasure to become practical. Irrelevance is stronger than erasability, which makes it too restrictive to be the only erasure mechanism.*

Chapter 3, in particular Section 3.1.2, shows concrete examples of programs that should run in linear time but in fact run in quadratic or exponential time. I also show that this problem is caused by extra data and code present in the program and that this data and code is not removable easily in current systems (Section 3.2).

The examples demonstrate that erasure enables a whole class of programming techniques (such as programming with dependent views) that would otherwise be too inefficient.

Irrelevance, implemented in languages like Agda or Zombie, is stronger than erasure (Section 2.1.7) and unless we need its effect on equality, erasure is a more flexible choice (Section 2.1.7.2) which admits easier inference (Section 7.6).

Erasure is feasible

- *There are algorithms that discover and erase non-computational data from such programs.*

  Chapter 4 on one hand, and Chapter 5 together with Chapter 6 on the other hand, give two different approaches to erasing non-computational data.
- *These algorithms are effective.*

Sections 4.6 and 9.1 demonstrate that in all programs introduced in Section 3.1.2, these algorithms are able to recover the expected linear runtime complexity, without any user-provided annotations.

- *These algorithms are reasonably efficient.* Theoretically, Section 6.5.1 shows that the time complexity of the presented inference algorithms, without erasure polymorphism and (the unimplemented) inference-speedup extensions, can reach at least $O(n^2)$ time in the size of the input program. Section 6.5.2 discusses several efficiency-improving extensions.

  Practically, Section 4.6.1.2, Figure 4.9 and Section 9.1, Figure 9.1d show that the net impact of erasure on total compilation time in the examined programs is actually *negative*: erasure saves more time than it takes, by reducing work in the subsequent stages, especially code generation and optimisation.

- *These algorithms are applicable to a real-world implementation of a practical dependently typed programming language.*

  For Chapter 4, I have a constructive proof – this approach is implemented in Idris and has been running on all programs for more than four years.

  The type-based erasure approach (Chapters 5, 6, and 7) is implemented in a small compiler, separately from Idris. However, the core language $\mathsf{TT}_\star$ was designed as an erasure-aware extension of Idris's TT, and its implementation shows that type-based erasure works well with features like I/O and foreign function interfaces (Section 7.5).

- *These algorithms are sound; they preserve the meanings of programs.*

  I prove several standard metatheoretic properties of $\mathsf{TT}_\star$ such as subject reduction, and, most importantly, soundness of erasure in the sense that it commutes with reduction (Section 5.7) in type- and erasure-correct programs. In doing so, I assume confluence of reduction.

  I have shown that the erasure inference algorithm given in Chapter 6 is sound and complete with respect to the typing rules of the calculus, and therefore it is safe to erase programs annotated by erasure inference.

9.4 Future work

Integration of QTT, $\mathsf{TT}_\star$, weakening, and irrelevance The most attractive direction of future research is integration of $\mathsf{TT}_\star$ (pattern matching, erasure inference) with QTT (Section 8.4.1.3), McBride-style weakening (Section 8.4.1.1), and possibly even irrelevance, especially since QTT has been chosen as the core calculus of Idris 2[3].

The two erasure levels E (erased) and R (unerased) are further differentiated into I (erased, irrelevant), E (erased, relevant) and L (unerased, linear/affine), R (unerased,

[3]Prototype implementation at https://github.com/edwinb/Blodwen/.

$\cdot$	I	E	L	R
I	I	I	I	I
E	I	E	E	E
L	I	E	L	R
R	I	E	R	R

$+$	I	E	L	R
I	I	E	L	R
E	E	E	L	R
L	L	L	R	R
R	R	R	R	R

$\leq$	I	E	L	R
I	$\leq$	$\leq$	$(\leq)$	$\leq$
E		$\leq$	$(\leq)$	$\leq$
L			$\leq$	$\leq$
R				$\leq$

FIGURE 9.5: Hypothetical integrated rig
The parenthesised weakening entries are present if L is affine, not present if L is linear.

unrestricted). The rig-based approach can accommodate all four modalities, together with a suitable weakening relation (Figure 9.5).

I and L are not just "erasure/security levels", but they affect type checking each in its own specific way. However, I have an experimental implementation of full I-E-L-R inference, which suggests that the approach discussed in Section 7.6.3 works for inference of irrelevance and erasure and is compatible with inference of linearity.

It remains to be seen how efficient it is and whether this way of integration of QTT with irrelevance, and inference is viable for practical programming in its full form.

Integration of $\mathsf{TT}_\star$ with Idris The simple version of erasure is currently implemented in Idris 1. Another possible goal would be inserting a $\mathsf{TT}_\star$ stage into the compiler pipeline of Idris 1. The main hurdle at the moment is that in Idris 1, information about forced patterns is not present yet in the surface language – and already forgotten in the elaborated intermediate representation. Modifications to the compiler will be needed to preserve this information.

Metatheory Metatheory of $\mathsf{TT}_\star$ needs more work. While I have proven subject reduction (Theorem 5.1) and that erasure commutes with reduction (Theorem 5.2), other metatheoretic properties would be useful to have, as sketched in Section 5.7.11. Especially confluence (Conjecture 5.1), on which correctness relies, would be useful.

Erasure to a typed calculus Mishra-Linger [ML08] erases to IPTS, which is a typed calculus. This might be a more systematic approach than erasure to an untyped calculus, since it preserves a lot of information that could be possibly useful in further compilation stages, or reasoning about the erasure semantics.

Efficiency of inference I have not put much effort into making erasure inference efficient beyond trying to avoid the most obvious of complexity issues.

This seems to work well in practice. Only recently, an implementation of the compiler of Idris 2 *in* Idris 1 started taking several seconds to solve erasure constraints. I implemented the indexed solver (Section 6.4), which reduced the erasure time back to insignificant values.

I am sure that besides this lowest-hanging fruit, there is much more space for optimisation in erasure inference, be it in generation of constraints, solving them, or other smarter approaches.

Case trees in the core calculus In retrospect, while pattern clauses in the core language are easier to reason about, case trees have other advantages (Section 7.2.1) and I think that a core language like $\mathsf{TT}_\star$ would be more useful with (baseline) case trees in as its pattern matching facility.

First-class erasure annotations As sketched in Section 7.3.1.10, it might be useful to have erasure annotations as first-class values in the calculus, similar to universe levels in Agda.

Sections 2.2.4.3 and 7.3.1.10 show that this is something that we can already have now, at the expense of some boilerplate code.

Erasure polymorphism For type families, Section 7.3.2 talks about a method that has not been implemented yet. It would be useful to find out how well it works in practice, how severe its limitations are (e.g. Section 7.3.2.5) and how efficient it is.

For functions, the approach with type signature duplication (Section 7.3.1) has not been tested at scale yet. While it seems to work well on small programs, it is unknown how expensive it would be if e.g. every reference to a function in a bigger program would be erasure-polymorphic.

Metatheory of extensions Extensions in Chapter 7 come without much theoretical justification. Especially for erasure polymorphism, adding metatheory would establish its trustworthiness and help understand it better. Similarly, the material on case trees in Section 7.2 should be formally tied to literature [GMM06].

Control flow analysis It might be interesting to investigate how control flow analysis could help erasure polymorphism inference. Purely type-based distinction of erasure variants seems to be too coarse (Section 7.3.2.5) and a control flow analysis could yield a more flexible and precise erasure polymorphism inference.

More precise complexity bounds Section 6.5.1 shows that there is a quadratic case for the erasure inference algorithm. Its true complexity is however unknown.

More test programs and benchmarks While the three running examples, binary numbers, palindrome checker, and RLE, illustrate the problem, they are relatively small and – especially the palindrome checker – somewhat contrived.

It would be useful to collect more programs, in the spirit of Haskell's nofib suite [Par93], that could be used to test the erasure mechanism at a larger scale.

Bibliography

[Abe04] Andreas Abel. "Termination checking with types". In: *RAIRO - Theoretical Informatics and Applications* 38.4 (2004), 277–319. DOI: 10.1051/ita:2004015.

[Abe11] Andreas Abel. "Irrelevance in Type Theory with a Heterogeneous Equality Judgement". In: *Foundations of Software Science and Computational Structures: 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings*. Ed. by Martin Hofmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 57–71. ISBN: 978-3-642-19805-2. DOI: 10.1007/978-3-642-19805-2_5. URL: https://doi.org/10.1007/978-3-642-19805-2_5.

[Abe17] Andreas Abel. *Irrelevance and resurrection in type signatures*. Agda Mailing List, 29 July 2017. 2017. URL: https://lists.chalmers.se/pipermail/agda/2017/009640.html.

[Abe98] Andreas Abel. *foetus – Termination Checker for Simple Functional Programs*. 1998.

[Agd14] Agda authors. *Agda Wiki: Irrelevance*. Accessed on 25 Feb 2015. 2014. URL: http://wiki.portal.chalmers.se/agda/pmwiki.php?n=ReferenceManual.Irrelevance.

[Agd17a] Agda authors. *Agda Wiki*. Accessed on 21 Sep 2017. 2017. URL: http://wiki.portal.chalmers.se/agda/pmwiki.php.

[Agd17b] Agda authors. *Agda Wiki: Magic With*. Accessed on 21 Sep 2017. 2017. URL: http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Docs.MagicWith.

[Aho+06] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[AP11] Andreas Abel and Brigitte Pientka. "Higher-Order Dynamic Pattern Unification for Dependent Types and Records". In: *Typed Lambda Calculi and Applications: 10th International Conference, TLCA 2011, Novi Sad, Serbia, June 1-3, 2011. Proceedings*. Ed. by Luke Ong. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 10–26. ISBN: 978-3-642-21691-6. DOI: 10.1007/978-3-642-21691-6_5. URL: https://doi.org/10.1007/978-3-642-21691-6_5.

[AS12] Andreas Abel and Gabriel Scherer. "On irrelevance and algorithmic equality in predicative type theory". In: *arXiv preprint arXiv:1203.4716* (2012).

[Atk18] Robert Atkey. "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*. Ed. by Anuj Dawar and Erich Grädel. ACM, 2018, pp. 56–65. DOI: 10.1145/3209108.3209189. URL: https://doi.org/10.1145/3209108.3209189.

[Aug85] Lennart Augustsson. "Compiling Pattern Matching". In: *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*. Nancy, France: Springer-Verlag New York, Inc., 1985, pp. 368–381. ISBN: 3-387-15975-4. URL: http://dl.acm.org/citation.cfm?id=5280.5303.

[Aug99] Lennart Augustsson. "Cayenne — A Language with Dependent Types". In: *Advanced Functional Programming: Third International School, AFP'98, Braga, Portugal, September 12-19, 1998, Revised Lectures*. Ed. by S. Doaitse Swierstra, José N. Oliveira, and Pedro R. Henriques. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 240–267. ISBN: 978-3-540-48506-3. DOI: 10.1007/10704973_6. URL: http://dx.doi.org/10.1007/10704973_6.

[AVW17] Andreas Abel, Andrea Vezzosi, and Theo Winterhalter. "Normalization by evaluation for sized dependent types". In: *Proceedings of the ACM on Programming Languages* 1.ICFP (2017), p. 33.

[Bar91] Henk Barendregt. "Introduction to generalized type systems". In: *Journal of functional programming* 1.2 (1991), pp. 125–154.

[BB08] Bruno Barras and Bruno Bernardo. "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. Budapest, Hungary: Springer-Verlag, 2008, pp. 365–379. ISBN: 3-540-78497-7, 978-3-540-78497-5. URL: http://dl.acm.org/citation.cfm?id=1792803.1792829.

[BC05] Ana Bove and Venanzio Capretta. "Modelling general recursion in type theory". In: *Mathematical Structures in Computer Science* 15.4 (2005), pp. 671–708. URL: http://www-sop.inria.fr/lemme/Venanzio.Capretta/publications/general_recursion.pdf.

[Ber+00] Stefano Berardi et al. "Type-based useless-code elimination for functional programs position paper". In: *Semantics, Applications, and Implementation of Program Generation*. Springer, 2000, pp. 172–189.

[BJP12] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. "Proofs for Free: Parametricity for Dependent Types". In: *J. Funct. Program.* 22.2 (Mar. 2012), pp. 107–152. ISSN: 0956-7968. DOI: 10.1017/S0956796812000056. URL: http://dx.doi.org/10.1017/S0956796812000056.

[BM13] Jean-Philippe Bernardy and Guilhem Moulin. "Type-theory in Color". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. Boston, Massachusetts, USA: ACM, 2013, pp. 61–72. ISBN: 978-1-4503-2326-0. DOI: 10.1145/2500365.2500577. URL: http://doi.acm.org/10.1145/2500365.2500577.

[BMM04] Edwin Brady, Conor McBride, and James McKinna. "Inductive Families Need Not Store Their Indices". In: *Types for Proofs and Programs, Torino, 2003*. Ed. by Stefano Berardi, Mario Coppo, and Ferrucio Damiani. Vol. 3085. LNCS. Springer-Verlag, 2004, pp. 115–129.

[Bra05] Edwin Brady. "Practical Implementation of a Dependently Typed Functional Programming Language". PhD thesis. 2005. URL: http://eb.host.cs.st-andrews.ac.uk/writings/thesis.pdf.

[Bra13] Edwin Brady. "Idris, a general-purpose dependently typed programming language: Design and implementation". In: *Journal of Functional Programming* 23.5 (2013), 552–593. DOI: 10.1017/S095679681300018X.

[Bra17] Edwin Brady. *Type-Driven Development with Idris*. Manning Publications Company, 2017. ISBN: 9781617293023. URL: https://books.google.de/books?id=eWzEjwEACAAJ.

[Bru+14] Aloïs Brunel et al. "A Core Quantitative Coeffect Calculus". In: *Programming Languages and Systems: 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. Ed. by Zhong Shao. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 351–370. ISBN: 978-3-642-54833-8. DOI: 10.1007/978-3-642-54833-8_19. URL: https://doi.org/10.1007/978-3-642-54833-8_19.

[Bru91] N.G. de Bruijn. "Telescopic mappings in typed lambda calculus". In: *Information and Computation* 91.2 (1991), pp. 189 –204. ISSN: 0890-5401. DOI: http://dx.doi.org/10.1016/0890-5401(91)90066-B. URL: http://www.sciencedirect.com/science/article/pii/089054019190066B.

[Cap01] Venanzio Capretta. "Certifying the fast Fourier transform with Coq". In: *TPHOLs*. Vol. 2152. Springer. 2001, pp. 154–168.

[Car88] Luca Cardelli. *Phase distinctions in type theory*. 1988.

[Cas14] Chris Casinghino. "Combining Proofs and Programs". PhD thesis. The University of Pennsylvania, 2014.

[CHI20] The CHICKEN project. *CHICKEN Scheme*. https://www.call-cc.org/. Accessed: 2020-04-25. 2020.

[Chl13] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. 2013.

[Coc17] Jesper Cockx. "Dependent Pattern Matching and Proof-Relevant Unification". PhD thesis. KU Leuven, 2017.

[Coq92] Thierry Coquand. "Pattern matching with dependent types". In: *Proceedings of the Workshop on Types for Proofs and Programs*. 1992, pp. 71–83.

[CP90] Thierry Coquand and Christine Paulin. "Inductively Defined Types". In: *Proceedings of the International Conference on Computer Logic*. COLOG '88. London, UK, UK: Springer-Verlag, 1990, pp. 50–66. ISBN: 3-540-52335-9. URL: http://dl.acm.org/citation.cfm?id=646125.758641.

[CSW14] Chris Casinghino, Vilhelm Sjöberg, and Stephanie 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. San Diego, California, USA: ACM, 2014, pp. 33–45. ISBN: 978-1-4503-2544-8. DOI: 10.1145/2535838.2535883. URL: http://doi.acm.org/10.1145/2535838.2535883.

[Den76] Dorothy E. Denning. "A Lattice Model of Secure Information Flow". In: *Commun. ACM* 19.5 (May 1976), pp. 236–243. ISSN: 0001-0782. DOI: 10.1145/360051.360056. URL: http://doi.acm.org/10.1145/360051.360056.

[DG84] William F. Dowling and Jean H. Gallier. "Linear-time algorithms for testing the satisfiability of propositional horn formulae". In: *The Journal of Logic Programming* 1.3 (1984), pp. 267 –284. ISSN: 0743-1066. DOI: http://dx.doi.org/10.1016/0743-1066(84)90014-1. URL: http://www.sciencedirect.com/science/article/pii/0743106684900141.

[Doc14] Robert 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.

[Dur+14] Zakir Durumeric et al. "The matter of heartbleed". In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. ACM. 2014, pp. 475–488.

[Dyb94] Peter Dybjer. "Inductive Families". In: *Formal Asp. Comput.* 6.4 (1994), pp. 440–465. DOI: 10.1007/BF01211308. URL: https://doi.org/10.1007/BF01211308.

[Eis16] Richard A Eisenberg. "Dependent types in Haskell: Theory and practice". PhD thesis. University of Pennsylvania, 2016.

[EJ00] Martin Erwig and Simon Peyton Jones. *Pattern Guards and Transformational Patterns*. 2000. DOI: 10.1016/S1571-0661(05)80540-7.

[FH01] Adam Fischbach and John Hannan. "Type Systems for Useless-Variable Elimination". In: *Programs as Data Objects: Second Symposium, PADO2001 Aarhus, Denmark, May 21–23, 2001 Proceedings*. Ed. by Olivier Danvy and Andrzej Filinski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 25–38. ISBN: 978-3-540-44978-2. DOI: 10.1007/3-540-44978-7_3. URL: https://doi.org/10.1007/3-540-44978-7_3.

[Gir72] Jean-Yves Girard. "Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur". PhD thesis. PhD thesis, Université Paris VII, 1972.

[GL88] Michael Gelfond and Vladimir Lifschitz. "The stable model semantics for logic programming." In: *ICLP/SLP*. Vol. 88. 1988, pp. 1070–1080.

[GM12] Adam Gundry and Conor McBride. "A tutorial implementation of dynamic pattern unification". In: *Draft* (2012). URL: http://adam.gundry.co.uk/pub/pattern-unify/.

[GM13] Adam Gundry and Conor McBride. "Phase Your Erasure". 2013. URL: https://personal.cis.strath.ac.uk/conor.mcbride/pub/phtt.pdf.

[GMM06] Healfdene Goguen, Conor McBride, and James McKinna. "Eliminating Dependent Pattern Matching". In: *Algebra, Meaning, and Computation: Essays dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*. Ed. by Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 521–540. ISBN: 978-3-540-35464-2. DOI: 10.1007/11780274_27. URL: http://dx.doi.org/10.1007/11780274_27.

[GS14] Dan R. Ghica and Alex I. Smith. "Bounded Linear Types in a Resource Semiring". In: *Programming Languages and Systems: 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. Ed. by Zhong Shao. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 331–350. ISBN: 978-3-642-54833-8. DOI: 10.1007/978-3-642-54833-8_18. URL: https://doi.org/10.1007/978-3-642-54833-8_18.

[Gun13] Adam Gundry. "Type Inference, Haskell and Dependent Types". PhD thesis. University of Strathclyde, 2013. URL: http://adam.gundry.co.uk/pub/thesis/.

[Har11] Robert Harper. *Boolean Blindness*. https://existentialtype.wordpress.com/2011/03/15/boolean-blindness/. Blog. 2011.

[Har16] Robert Harper. *Practical foundations for programming languages*. Cambridge University Press, 2016.

[How80] William A. Howard. "The formulas-as-types notion of construction". In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Ed. by J. P. Seldin and J. R. Hindley. Reprint of 1969 article. Academic Press, 1980, pp. 479–490.

[Hur95] Antonius JC Hurkens. "A simplification of Girard's paradox". In: *International Conference on Typed Lambda Calculi and Applications*. Springer. 1995, pp. 266–278.

[JM02] Simon Peyton Jones and Simon Marlow. "Secrets of the glasgow haskell compiler inliner". In: *Journal of Functional Programming* 12.4-5 (2002), pp. 393–434.

[Jon03] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[Lei14a] Jonathan Leivent. *Erasable relevance*. https://github.com/jonleivent/mindless-coding/blob/fd2d662381aa7a805c73ac2bfeb1f1cadcfea47a/erasable_relevance.v, accessed on 2015-02-28. 2014.

[Lei14b] Jonathan Leivent. *Mindless Coding*. https://github.com/jonleivent/mindless-coding, accessed on 2015-02-28. 2014.

[Let03] Pierre Letouzey. "A New Extraction for Coq". English. In: *Types for Proofs and Programs*. Ed. by Herman Geuvers and Freek Wiedijk. Vol. 2646. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 200–219. ISBN: 978-3-540-14031-3. DOI: 10.1007/3-540-39185-1_12. URL: http://dx.doi.org/10.1007/3-540-39185-1_12.

[Let04] P. Letouzey. "Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq". PhD thesis. Université Paris-Sud, July 2004.

[Let08] P. Letouzey. "Coq Extraction, an Overview". In: *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*. Ed. by A. Beckmann, C. Dimitracopoulos, and B. Löve. Vol. 5028. Lecture Notes in Computer Science. Springer-Verlag, 2008. URL: http://www.pps.univ-paris-diderot.fr/~letouzey/download/letouzey_extr_cie08.pdf.

[LJBA01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. "The size-change principle for program termination". In: *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '01. London, United Kingdom: ACM, 2001, pp. 81–92. ISBN: 1-58113-336-7. DOI: 10.1145/360204.360210. URL: http://doi.acm.org/10.1145/360204.360210.

[LS05] Pierre Letouzey and Bas Spitters. "Implicit and noncomputational arguments using monads". In: 2005.

[LT93] Nancy G Leveson and Clark S Turner. "An investigation of the Therac-25 accidents". In: *Computer* 26.7 (1993), pp. 18–41.

[Luo94] Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. 1994.

[Mar08] Luc Maranget. "Compiling Pattern Matching to Good Decision Trees". In: *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*. ML '08. Victoria, BC, Canada: ACM, 2008, pp. 35–46. ISBN: 978-1-60558-062-3. DOI: 10.1145/1411304.1411311. URL: http://doi.acm.org/10.1145/1411304.1411311.

[McB00] Conor McBride. "Dependently typed functional programs and their proofs". PhD thesis. University of Edinburgh. College of Science and Engineering. School of Informatics., 2000.

[McB02] Conor McBride. "Elimination with a Motive". In: *Types for Proofs and Programs (Proceedings of the International Workshop, TYPES'00)*. Ed. by Paul Callaghan et al. Vol. 2277. LNCS. Springer-Verlag, 2002.

[McB03] Conor McBride. "First-order unification by structural recursion". In: *Journal of functional programming* 13.6 (2003), pp. 1061–1075.

[McB05] Conor McBride. "Epigram: Practical Programming with Dependent Types". In: *Advanced Functional Programming*. Ed. by Varmo Vene and Tarmo Uustalu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 130–170. ISBN: 978-3-540-31872-9.

[McB12] Conor McBride. *Hindley-Milner Culture*. https://stackoverflow.com/a/13241158. 2012.

[McB14a] Conor McBride. *In A World*. Draft. 2014. URL: https://personal.cis.strath.ac.uk/conor.mcbride/pub/InAWorld.pdf.

[McB14b] Conor McBride. *worlds*. Blog post. 2014. URL: https://pigworker.wordpress.com/2014/12/31/worlds/.

[McB14c] Conor Thomas McBride. "How to keep your neighbours in order". In: *ACM SIGPLAN Notices*. Vol. 49. 9. ACM. 2014, pp. 297–309.

[McB15] Conor McBride. *Type Inference Needs Revolution*. http://staff.computing.dundee.ac.uk/frantisekfarka/tiap/#conor. 2015.

[McB16] Conor McBride. "I got plenty o'nuttin'". In: *A List of Successes That Can Change the World*. Springer, 2016, pp. 207–233.

[McB99] Conor McBride. "Dependently Typed Functional Programs and their Proofs". Available from http://www.lfcs.informatics.ed.ac.uk/reports/00/ECS-LFCS-00-419/. PhD thesis. University of Edinburgh, 1999.

[Mid12] Jan Midtgaard. "Control-flow analysis of functional programs". In: *ACM Computing Surveys (CSUR)* 44.3 (2012), p. 10.

[Miq01] Alexandre Miquel. "The Implicit Calculus of Constructions". In: *TLCA*. 2001, pp. 344–359. DOI: 10.1007/3-540-45413-6_27. URL: https://doi.org/10.1007/3-540-45413-6_27.

[ML08] Richard Nathan Mishra-Linger. *Irrelevance, Polymorphism, and Erasure in Type Theory*. 2008. URL: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=0B58D3AF3E6F156739E8A4D55FA049BD?doi=10.1.1.154.5619&rep=rep1&type=pdf.

[MLS84] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*. Vol. 9. Bibliopolis Napoli, 1984.

[MM04] Conor McBride and James McKinna. "The view from the left". In: *Journal of Functional Programming* 14.1 (2004), pp. 69–111.

[Mos+01] Matthew W Moskewicz et al. "Chaff: Engineering an efficient SAT solver". In: *Proceedings of the 38th annual Design Automation Conference*. ACM. 2001, pp. 530–535.

[MPC97] Catherine C McGeoch, Doina Precup, and Paul R Cohen. "How to find big-oh in your data set (and how not to)". In: *Advances in Intelligent Data Analysis. Reasoning about Data: Second International Symposium, IDA-97, London, UK, August 1997. Proceedings*. Springer. 1997, p. 41.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN: 1-55860-320-4.

[Myc80] Alan Mycroft. "The theory and practice of transforming call-by-need into call-by-value". In: *International symposium on programming*. Springer. 1980, pp. 269–281.

[Nor07] Ulf Norell. "Towards a practical programming language based on dependent type theory". PhD thesis. Chalmers University of Technology, 2007. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.65.7934&rep=rep1&type=pdf.

[NVD17] Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. "Parametric Quantifiers for Dependent Type Theory". In: *Proceedings of the 22th ACM SIGPLAN International Confirence on Functional Programming*. ICFP '17. 2017.

[Par93] Will Partain. "The nofib Benchmark Suite of Haskell Programs". In: *Functional Programming, Glasgow 1992: Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, 6–8 July 1992*. Ed. by John Launchbury and Patrick Sansom. London: Springer London, 1993, pp. 195–202. ISBN: 978-1-4471-3215-8. DOI: 10.1007/978-1-4471-3215-8_17. URL: https://doi.org/10.1007/978-1-4471-3215-8_17.

[Pau89] Christine 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*. 1989, pp. 89–104. DOI: 10.1145/75277.75285. URL: http://doi.acm.org/10.1145/75277.75285.

[Pfe01] Frank Pfenning. "Intensionality, extensionality, and proof irrelevance in modal type theory". In: *Logic in Computer Science, 2001. Proceedings. 16th Annual IEEE Symposium on*. IEEE. 2001, pp. 221–230.

[Pie01] Brigitte Pientka. "Termination and reduction checking for higher-order logic programs". In: *IJCAR*. Vol. 2083. Springer. 2001, pp. 401–415.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002. ISBN: 0262162091, 9780262162098.

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1987. ISBN: 013453333X.

[PJW93] Simon L Peyton Jones and Philip Wadler. "Imperative functional programming". In: *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1993, pp. 71–84.

[POM14] Tomas Petricek, Dominic Orchard, and Alan Mycroft. "Coeffects: A Calculus of Context-dependent Computation". In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. Gothenburg, Sweden: ACM, 2014, pp. 123–135. ISBN: 978-1-4503-2873-9. DOI: 10.1145/2628136.2628160. URL: http://doi.acm.org/10.1145/2628136.2628160.

[Rey83] John C Reynolds. "Types, abstraction and parametric polymorphism". In: (1983).

[SF00] Peter Sanders and Rudolf Fleischer. "Asymptotic complexity from experiments? A case study for randomized algorithms". In: *International Workshop on Algorithm Engineering*. Springer. 2000, pp. 135–146.

[Shi91] Olin Shivers. "Control-flow analysis of higher-order languages". PhD thesis. Carnegie-Mellon University, 1991.

[Sjö15] Vilhelm Sjöberg. "A Dependently Typed Language with Nontermination". PhD thesis. University of Pennsylvania, 2015.

[SM03] Andrei Sabelfeld and Andrew C Myers. "Language-based information-flow security". In: *IEEE Journal on selected areas in communications* 21.1 (2003), pp. 5–19.

[SP03] Carsten Schürmann and Frank Pfenning. "A coverage checking algorithm for LF". In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2003, pp. 120–135.

[SSW10] Tim Sheard, Aaron Stump, and Stephanie Weirich. "Language-based verification will change the world". In: *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM. 2010, pp. 343–348.

[Tah04] Walid Taha. "A gentle introduction to multi-stage programming". In: *Domain-Specific Program Generation*. Springer, 2004, pp. 30–50.

[The04] The Coq development team. *The Coq proof assistant reference manual*. Version 8.0. LogiCal Project. 2004. URL: http://coq.inria.fr.

[VEK76] Maarten H Van Emden and Robert A Kowalski. "The semantics of predicate logic as a programming language". In: *Journal of the ACM (JACM)* 23.4 (1976), pp. 733–742.

[VS97] Dennis Volpano and Geoffrey Smith. "A type-based approach to program security". In: *TAPSOFT '97: Theory and Practice of Software Development: 7th International Joint Conference CAAP/FASE Lille, France, April 14–18, 1997 Proceedings*. Ed. by Michel Bidoit and Max Dauchet. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 607–621. ISBN: 978-3-540-68517-3. DOI: 10.1007/BFb0030629. URL: https://doi.org/10.1007/BFb0030629.

[Wad00] Philip Wadler. "Proofs are programs: 19th century logic and 21st century computing". In: (2000).

[Wad15] Philip Wadler. "Propositions As Types". In: *Commun. ACM* 58.12 (Nov. 2015), pp. 75–84. ISSN: 0001-0782. DOI: 10.1145/2699407. URL: http://doi.acm.org/10.1145/2699407.

[Wad87a] Phil Wadler. "Efficient Compilation of Pattern Matching". In: *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1987. ISBN: 013453333X.

[Wad87b] Philip Wadler. "Views: A way for pattern matching to cohabit with data abstraction". In: *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM. 1987, pp. 307–313.

[Wad89] Philip Wadler. "Theorems for free!" In: *Proceedings of the fourth international conference on Functional programming languages and computer architecture*. ACM. 1989, pp. 347–359.

[Wei+17] Stephanie Weirich et al. "A Specification for Dependent Types in Haskell". In: *Proc. ACM Program. Lang.* 1.ICFP (Aug. 2017), 31:1–31:29. ISSN: 2475-1421. DOI: 10.1145/3110275. URL: http://doi.acm.org/10.1145/3110275.

[Wei81] Mark Weiser. "Program slicing". In: *Proceedings of the 5th international conference on Software engineering*. IEEE Press. 1981, pp. 439–449.

[WS99] Mitchell Wand and Igor Siveroni. "Constraint Systems for Useless Variable Elimination". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '99. San Antonio, Texas, USA: ACM, 1999, pp. 291–302. ISBN: 1-58113-095-3. DOI: 10.1145/292540.292567. URL: http://doi.acm.org/10.1145/292540.292567.

[ZBM99] Richard W. Zurek, Joseph M. Boyce, and John B. McNamee. *Mars Climate Orbiter*. National Aeronautics and Space Administration website, https://nssdc.gsfc.nasa.gov/nmc/spacecraftDisplay.do?id=1998-073A. Accessed on 2017-04-05. 1999.