# Dependent types

Jeremy Yallop

jeremy.yallop@cl.cam.ac.uk

## Origins

## The Curry-Howard correspondence

Correspondence between simply-typed language and propositional logics:

$$\begin{aligned} A \to B &\quad\simeq\quad A \supset B && \text{(functions and implications)} \\ A \times B &\quad\simeq\quad A \wedge B && \text{(products and conjunctions)} \\ A + B &\quad\simeq\quad A \wedge B && \text{(sums and disjunctions)} \end{aligned}$$

Correspondence between dependently-typed languages and predicate logics:

$$\begin{aligned} (x : A) \to B &\quad\simeq\quad \forall(x : A).B && \text{(functions and universal quantification)} \\ \Sigma(x : A).B &\quad\simeq\quad \exists A \wedge B && \text{(dependent pairs and existential quantification)} \end{aligned}$$

How should we **start** to design a dependently-typed language**?**

- Foundation for constructive mathematics (Martin-Löf Type Theory)
- Lambda calculus with fancy types (Calculus of Constructions)

## Equalities

With dependent types we can form **types from terms**.
Parameterise $B$ by a term of type $A$:

$$(x : A) \to B(x)$$

**Key Q**: when are two types equal? (essential for type checking!)

Is $B((\lambda x.x)Nat)$ equal to $B(Nat)$?

Determining equality typically requires **normalization** (i.e. computation).

(Separate question: what equalities can we **prove**?)

## Inductive families

## Inductive families: basics

Inductive families support **indexing data types by terms**:

an inductive family, Vect:

```
data Vect : Nat → Type → Type where
  Nil  : Vect Z a
  _::_ : a → Vect n a → Vect (S n) a
```

a function _++_ over Vect:

```
_++_ : Vect m a → Vect n a → Vect (m + n) a
_++_    Nil     ys = ys
_++_ (x :: xs) ys = x :: (xs ++ ys)
```

the full type of _++_:

```
_++_ : {a : Type} → {m : Nat} → {n : Nat} →
       Vect m a → Vect n a → Vect (m + n) a
```

# Inductive families and pattern matching

Matching on one value may reveal something about another.

**Example**:

```
_++_ : Vect m a → Vect n a → Vect (m + n) a
_++_    Nil     ys = ys
_++_ (x :: xs) ys = x :: (xs ++ ys)
```

1. Matching the first vector tells us that `m` is `Z`
2. `Vect (Z + n) a` ⇝ `Vect n a`
3. so `ys` has the appropriate type in the first branch

**Example**:

```
zip : Vect n a → Vect n b → Vect n (a,b)
zip    Nil     ys = ?
```

1. Matching the first vector tells us that `n` is `Z`
2. so the type of `ys` is `Vect Z b`
3. and so `Nil` is the only possible constructor for `ys`

## Inductive families and termination

**Ideally**: all functions terminate. *(Why?)*)

**Problem**: termination is undecidable, so we must approximate syntactically

**Question**: what to do with functions that are not structurally decreasing?

structurally decreasing:

```
length : [a] → Int
length []     = 0
length (x:xs) = 1 + length xs
```

not (obviously)
structurally decreasing:

```
quicksort :: [N] → [N]
quicksort [] = []
quicksort (x:xs) = quicksort (filter (<  x) xs) ++
                 x : quicksort (filter (>= x) xs).
```

# Inductive families and erasure

**Problem**: computationally-unnecessary code in elaborated programs

**Idea**: infer which parts can be erased to improve run-time performance

vector append:

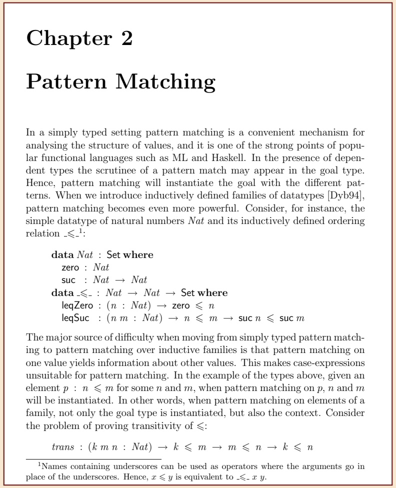
```
_++_ : Vect m a → Vect n a → Vect (m + n) a
_++_    Nil     ys = ys
_++_ (x :: xs) ys = x :: (xs ++ ys)
```

vector append, elaborated:

```
_++_ : (a : Type) → (m : Nat) → (n : Nat) →
       Vect m a → Vect n a → Vect (m + n) a
_++_ a   Z   n   Nil              ys = ys
_++_ a (S k) n ((::) a k x xs) ys =
   (::) a (k+n) x (_++_ a k n xs ys)
```

## Reading

# Reading 1: pattern matching



# Chapter 2

# Pattern Matching

In a simply typed setting pattern matching is a convenient mechanism for analysing the structure of values, and it is one of the strong points of popular functional languages such as ML and Haskell. In the presence of dependent types the scrutinee of a pattern match may appear in the goal type. Hence, pattern matching will instantiate the goal with the different patterns. When we introduce inductively defined families of datatypes [Dyb94], pattern matching becomes even more powerful. Consider, for instance, the simple datatype of natural numbers *Nat* and its inductively defined ordering relation $\_\leqslant\_$[1]:

```
data Nat : Set where
  zero : Nat
  suc  : Nat → Nat
data _≤_ : Nat → Nat → Set where
  leqZero : (n : Nat) → zero ≤ n
  leqSuc  : (n m : Nat) → n ≤ m → suc n ≤ suc m
```

The major source of difficulty when moving from simply typed pattern matching to pattern matching over inductive families is that pattern matching on one value yields information about other values. This makes case-expressions unsuitable for pattern matching. In the example of the types above, given an element $p : n \leqslant m$ for some $n$ and $m$, when pattern matching on $p$, $n$ and $m$ will be instantiated. In other words, when pattern matching on elements of a family, not only the goal type is instantiated, but also the context. Consider the problem of proving transitivity of $\leqslant$:

$$trans : (k\ m\ n : Nat) \to k \leqslant m \to m \leqslant n \to k \leqslant n$$

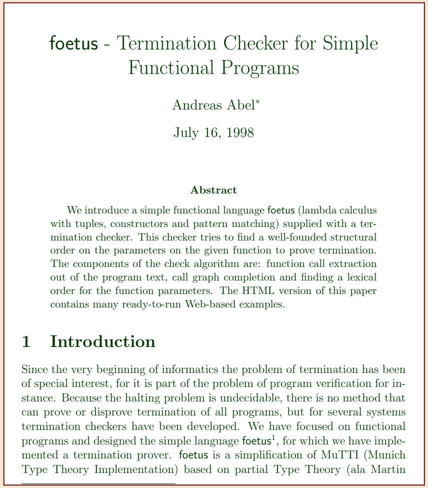
[1]Names containing underscores can be used as operators where the arguments go in place of the underscores. Hence, $x \leqslant y$ is equivalent to $\_\leqslant\_\ x\ y$.

"[W]e present the type checking algorithm for systems of pattern match equations. Contrary to previous work we allow equations to overlap and prioritise the rules from top to bottom [...]

"In many previous presentations coverage checking is undecidable [...] To solve this problem we [...] require programs to contain explicit dismissal of elements in empty types [...]

"The *with* construct, introduced by McBride and McKinna, allows analysis of intermediate results to be performed on the left hand side of a function definition rather than on the right hand side."

## Reading 2: termination checking



foetus - Termination Checker for Simple Functional Programs

Andreas Abel*

July 16, 1998

**Abstract**

We introduce a simple functional language foetus (lambda calculus with tuples, constructors and pattern matching) supplied with a termination checker. This checker tries to find a well-founded structural order on the parameters on the given function to prove termination. The components of the check algorithm are: function call extraction out of the program text, call graph completion and finding a lexical order for the function parameters. The HTML version of this paper contains many ready-to-run Web-based examples.

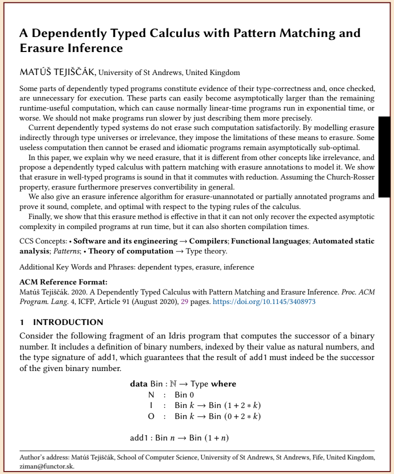
**1 Introduction**

Since the very beginning of informatics the problem of termination has been of special interest, for it is part of the problem of program verification for instance. Because the halting problem is undecidable, there is no method that can prove or disprove termination of all programs, but for several systems termination checkers have been developed. We have focused on functional programs and designed the simple language foetus[1], for which we have implemented a termination prover. foetus is a simplification of MuTTI (Munich Type Theory Implementation) based on partial Type Theory (ala Martin

"We introduce a simple functional language foetus (lambda calculus with tuples, constructors and pattern matching) supplied with a termination checker. This checker tries to find a well-founded structural order on the parameters on the given function to prove termination […]

"To prove the termination of a functional program there has to be a well founded order on the product of the function parameters such that the arguments in each recursive call are smaller than the corresponding input regarding this order."

# Reading 3: erasure inference



**A Dependently Typed Calculus with Pattern Matching and Erasure Inference**

MATÚŠ TEJIŠČÁK, University of St Andrews, United Kingdom

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

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

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

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

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

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

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

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

**1 INTRODUCTION**

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

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

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

"Some parts of dependently typed programs constitute evidence of their type-correctness and, once checked, are unnecessary for execution. These parts can easily become asymptotically larger than the remaining runtime-useful computation, which can cause normally linear-time programs run in exponential time, or worse [...]

"We show that erasure in well-typed programs is sound in that it commutes with reduction. Assuming the Church-Rosser property, erasure furthermore preserves convertibility in general."

## Writing suggestions

**Types**
How do compilers for dependently typed languages make use of types?

**Termination**
Is there a connection between erasure and accessibility predicates?

**Efficiency**
Are dependent types an impediment or an aid to efficiency?

**Decidability**
What undecidable questions arise in compilation with dependent types?

**Usability**
Do dependent types aid or impede usability?
Are inductive families an improvement over “recursive families”?