Phase Your Compilation

Constantine Theocharis

(λ,TRIPLE,2026)

What will it take to use dependent type theory for practical programming?

What is practical programming?

Can mean many things. I take it to mean:

Programming that is *abstract* when we want, *concrete* when we want.

- Dependent type theory certainly handles the abstract.
- But not so much the concrete.
  - This talk.

In dependent type theory, types can depend on values.

```
repeat : A → (n : Nat) → Vect A n
repeat x 0 = []
repeat x (1 + n) = [x, ..repeat x n]
```

We can express invariants directly in the type system.

- Eliminate out of bounds errors
- Express ‘arbitrary data refinement’

Nat is the type of natural numbers.

```
data Nat : Type where
  zero : Nat
  succ : Nat → Nat
```

Fin n is the type of natural numbers *less than* n.

```
data Fin : (n : Nat) → Type where
  fzero : ∀ k . Fin (succ k)
  fsucc : ∀ k . Fin k → Fin (succ k)
```

`Vect A n` is the type of lists of `A` with `n` elements.

```
data Vect A : Nat → Type where
  nil : Vect A zero
  cons : A → Vect A n → Vect A (succ n)
```

`0`	`≜ zero`	`0f`	`≜ fzero`	`[]`	`≜ nil`
`1 + n`	`≜ succ n`	`1f + i`	`≜ fsucc i`	`[x, ..xs]`	`≜ cons x xs`

With these, we can write a *total* indexing function:

```
lookup : Vect A n → Fin n → A
lookup [x, ..xs] 0f = x
lookup [x, ..xs] (1f + i) = lookup xs i
```

No need for bounds check because **the types guarantee the index is in range**.

Clash between worlds

Type theory is designed around mathematical meaning;

- Sets and functions
- Propositions and logical entailment proofs
- Various kinds of spaces & continuous maps
- …

However, we write programs for **computers** to run, which are **messy**;

- cost, memory allocation, cache locality, time/space complexity, finite bounds, …

How can we **augment** dependent type theory to get more **control** over these messy details?

Erasure

Mark data that has no runtime significance.

```
lookup : {0 n : Nat} → Vect A n → Fin n → A
```

- `n` exists only for the ‘mathematical meaning’, and is never used at runtime, so can be safely erased.

Additional constraints **superimposed** on top of DTT

Erasure exists in Idris 2 and Agda.

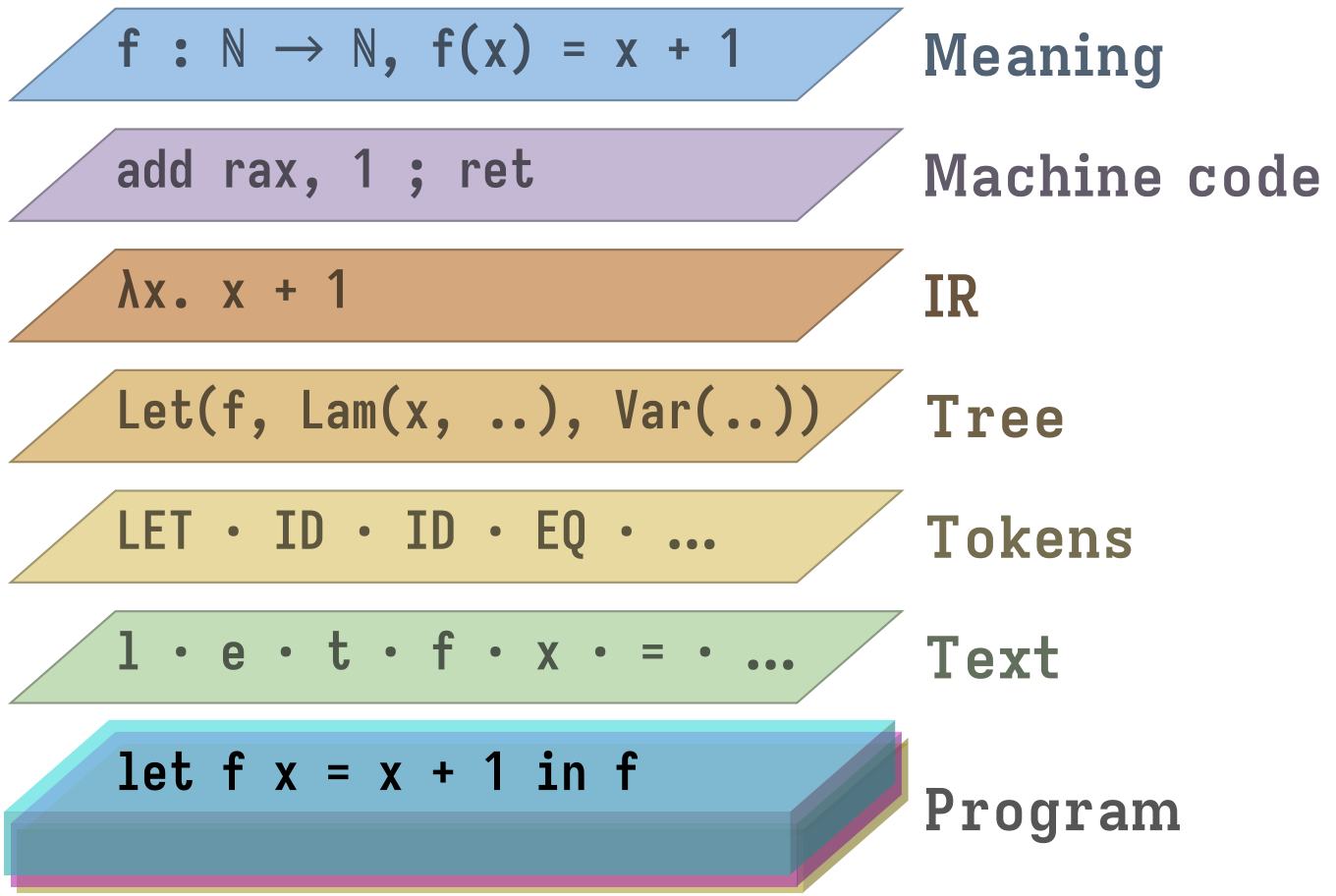
Can we go further?

Erasure handles: “this data doesn’t exist at runtime.”

But what about: “this data exists, but I want to control **its representation**.”

What does it mean to **superimpose** constraints onto a type theory?

Phase distinctions


f : ℕ → ℕ, f(x) = x + 1
Meaning
add rax, 1 ; ret
Machine code
λx. x + 1
IR
Let(f, Lam(x, ..), Var(..))
Tree
LET · ID · ID · EQ · …
Tokens
l · e · t · f · x · = · …
Text
let f x = x + 1 in f
Program

Phase distinctions let us reason about layers of information **in the type system**.

Work on phase distinctions

- **Compile-time vs runtime:** Cardelli 1988
- **Static vs dynamic**: Harper & Moggi 1989 — ML modules
- **Type theory in colour:** Bernardy & Moulin 2011 — annotations on binders distinguish phases
- **Quantitative type theory:** McBride 2014, Atkey 2018 — semiring annotations track resource usage (Idris 2)
- **Synthetic phase distinctions:** Sterling 2022

Formally,

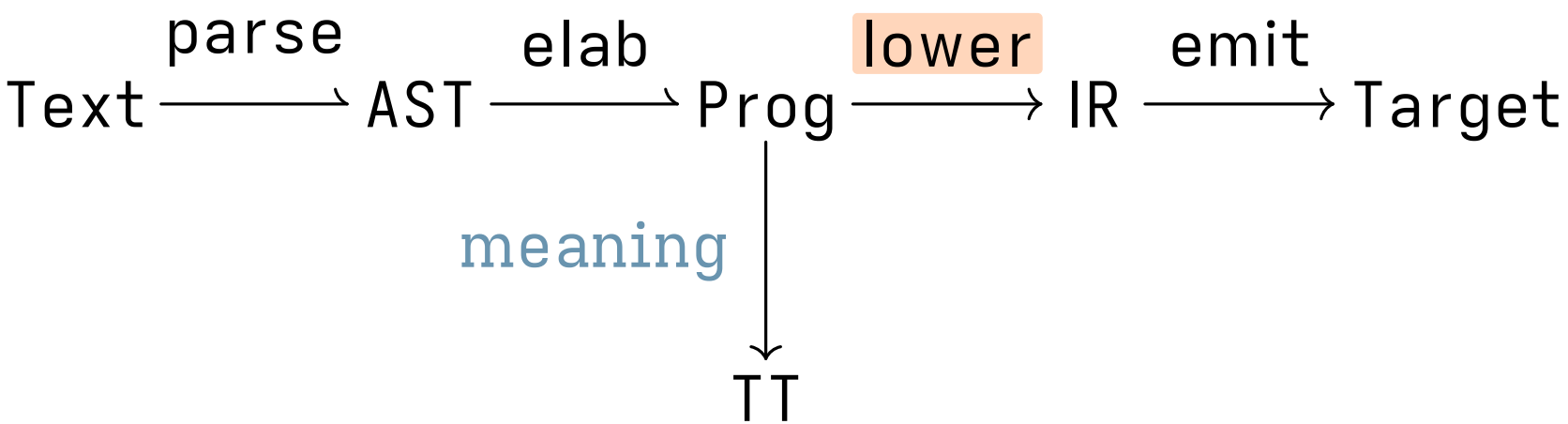
A phase distinction starts off as an **internalisation** of a translation between languages.

Given two languages M and N, and a translation

$$T : M \to N$$

???

we can form the ‘glued’ language $\mathsf{Gl}(T)$. It contains both M and N, and the action of T on M.


parse
elab
lower
emit
Text
AST
Prog
IR
Target
meaning
TT

(An) Intermediate Representation

Lowering removes erased data and **applies some optimisations**, preserving the computational content of the program.

$\alpha \Rightarrow \beta$	$\lambda\ x\ .\ t \mid t\ u$	functions
$\alpha \times \beta$	$\mathtt{pair}\ a\ b \mid \mathtt{fst}\ p \mid \mathtt{snd}\ p$	pairs
$\mathbf{unit}, \mathbf{nat}, \mathbf{list}\ \alpha$	$\mathtt{tt} \mid \mathtt{0} \mid \mathtt{1} + t \mid \mathtt{rec} \mid \ldots$	primitives

```
append : {0 m n: Nat} → Vect A m → Vect A n → Vect A (m + n)
append [] ys = ys
append [x, ..xs] ys = [x, ..append xs ys]
```

After lowering, erased indices disappear:

```
append : list α → list α → list α
append [] ys = ys
append [x, ..xs] ys = [x, ..append xs ys]
```

Internalising lowering

Let's extend Prog by the map $|-| : \mathsf{Prog} \to \mathsf{IR}$:

$$
\begin{aligned}
|\mathbf{Nat}| &= \mathbf{nat} \\
|\mathbf{Fin}\ k| &= \mathbf{nat} \\
|\mathbf{Vect}\ A\ n| &= \mathbf{list}\ |A| \\
|(x : A) \to B\ x| &= |A| \to |B\ \star| \\
|(0\ x : A) \to B\ x| &= |B\ \star|
\end{aligned}
$$

- Erased terms vanish: $|e| = \star$ if e is erased.
- Runtime terms survive, get stripped of dependency.

Being more precise

Prog : dependent type theory with CBPV, static universes,
erasure with erased β,η.

IR : simply-typed lambda calculus (no β,η) with CBPV+exceptions.

$|-|$: 'forgetful' map from Prog to IR

Why CBPV (call-by-push-value)?

- **Allows effects in IR, like exceptions.**
- Controlling computations vs values is useful for other things: e.g. closure control (Kovács 2024).

What are static universes?

- Universe indexed by representations

$$|\texttt{Type}\ \alpha| = \texttt{unit}$$
$$|\texttt{El}\ \{\alpha\}\ A| = \alpha$$

- Types are representation-monomorphic.
  - Runtime dependent types must be uniform in α.
  - Compile-time dependent types are unrestricted.
  - Can go further

We have access to lowering | - | **in the type system**.

Can enforce **constraints** based on lowering behaviour.

Specialisation types

Given a type `A` with representation `α`, **`Spec`** `A c` is the type of elements of `A` that are "equivalent" to `c : α`.

```
Spec Nat 42
```

- any natural number whose IR is **equivalent** to `42 : nat`

```
Spec ({0 n : Nat} → Fin n → Fin (1 + n)) (λ x. x)
```

- any map `Fin n → Fin (1 + n)` whose IR is **equivalent** to the identity `λ x. x : nat → nat`.

Using Spec

1. Given some a : A, where |a| is ‘equivalent’ to c through a proof p : |a| ~ c, we can specialise a to representation c by

$$\text{spec } a \text{ by } p : \textbf{Spec } A\ c$$

2. Given some b : **Spec** A c, we can recover an A by

$$\text{unspec } b : A$$

   where |unspec b| = c.

Weakening on `Fin`

We can write a map

```
weaken : {0 n : Nat} → Fin n → Fin (1 + n)
weaken 0f = 0f
weaken (1f + i) = 1f + (weaken i)
```

Because the `Fin` index is erased, this is equivalent to the identity function on `nat`.

So if p : |weaken| ~ λ x . x, then

```
spec weaken by p
  : Spec ({0 n : Nat} → Fin n → Fin (1 + n)) (λ x . x)
```

therefore

```
let weaken-optim = unspec (spec weaken by p)
  : {0 n : Nat} → Fin n → Fin (1 + n)
```

Optimisations are transparent to erasure (under `0`):

```
refl : weaken = weaken-optim
```

But they extract to different things (under `| - |`)

```
|weaken|       = λ x . rec 0 (+1) x
|weaken-optim| = λ x . x
```

We need a proof of `p : |weaken| ~ λ x . x`.

By lowering rules this becomes

$$p : (\lambda\ x\ .\ \mathrm{rec}\ 0\ (+1)\ x) \sim \lambda\ x\ .\ x$$

`_ ~ _ : α → α → Prop` is a relation on IR code that lives in the **meta** level (2LTT). It relates equivalent terms.

Contextual equivalence _ ~ _

- A relation defined inductively on the IR syntax
- Includes β, η rules for functions, computation rules for pairs, and natural numbers.
- Might include parametricity rules too, e.g. uniqueness of recursors

In the end, the lowered program must behave **observably the same** but perhaps with **different runtime characteristics**.

```
(λ x . rec 0 (+1) x) ~ λ x . x
= rec 0 (+1) x ~ x                    (λ-cong)
= x ~ x                               (nat-rec-η)
```

In the language:

```
spec weaken by λ-cong nat-rec-η
  : Spec ({0 n : Nat} → Fin n → Fin (1 + n)) (λ x . x)
```

Adding primitives

- Assuming `nat` is later compiled to a binary number:

```
_+_ : nat → nat → nat
_*_ : nat → nat → nat
_-1 : nat → nat
_-_ : nat → nat → nat
```

Each of these are equivalent to recursive variants, **but not equal**. e.g. `+-def : x + y ~ rec y (+1) x`

Detecting primitive operations

```
add : Nat → Nat → Nat
add 0 y = y
add (1 + x) y = 1 + (add x y)
add x y @spec x + y by +-def
```

```
mul : Nat → Nat → Nat
mul 0 y = 0
mul (1 + x) y = add y (mul x y)
mul x y @spec x * y by *-def
```

```
pred : (n : Nat) → {0 m : Nat} → n = 1 + m → Nat
pred (1 + n) refl = n
pred n @spec n - 1 by -1-def
```

```
shift : (k : Nat) → Fin n → Fin (k + n)
shift 0 i = i
shift (1 + k) i = 1f + (shift k i)
shift k i @spec k + i by +-def
```

We can find most of these proofs automatically!

Identity functions

Any map on inductive data which only **shifts a type-level index** is equivalent to the identity function at runtime.

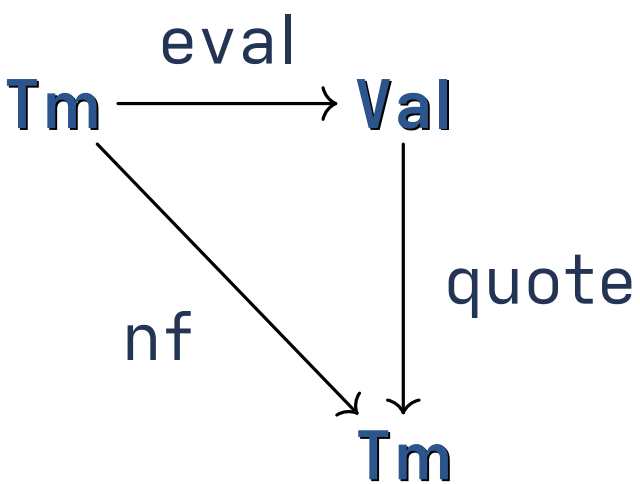
This happens all the time with dependent types!

```
last : (n : Nat) → Fin (1 + n)
last 0 = 0f
last (1 + k) = 1f + (last k)
last @id
```

Example: normalisation by evaluation

Normalisation by evaluation

- A way to normalise λ calculus terms.
- **Evaluate** a term to a value, then **reify** the value back.



de Brujin levels & indices

```
data Lvl : Nat → Type nat
where
  lz : Lvl (1 + n)
  ls : Lvl n → Lvl (1 + n)
```

```
data Idx : Nat → Type nat
where
  iz : Idx (1 + n)
  is : Idx n → Idx (1 + n)
```

λ v.	λ w.	λ x.	λ y.	λ z.	t
0	1	2	3	4	→ **Lvl** 5
4	3	2	1	0	← **Idx** 5

Terms & values

```
data Tm : Nat → Type _ where
  lam : Tm (1 + n) → Tm n
  app : Tm n → Tm n → Tm n
  var : Idx n → Tm n

data Val : Nat → Type _ where
  vspine : Lvl n → SnocList (Val n) → Val n
  vlam : Vect (Val n) m → Tm (1 + m) → Val n
```

Evaluation

```
lookup : Vect (Val n) m → Idx m → Val n
lookup [x, ..xs] iz = x
lookup [x, ..xs] (is n) = lookup xs n

eval : Vect (Val n) m → Tm m → Val n
eval env (lam t) = vlam env t
eval env (app t u) with eval env t | eval env u
... | vlam env' t' | u' = eval [u', ..env'] t'
... | vspine x sp  | u' = vspine x [..sp, u']
eval env (var i) = lookup env i
```

Reification

```
extend : {n: Nat} → Vect (Val n) m → Vect (Val (1 + n)) (1 + m)
extend env = [lastLvl n, ..map weak env]

quote : (n : Nat) → Val n → Tm n
quote n (vspine x ys) = fold app (lvlToIdx n x)
quote n (vlam env f) = lam (quote (1 + n) (eval (extend env) f))
```

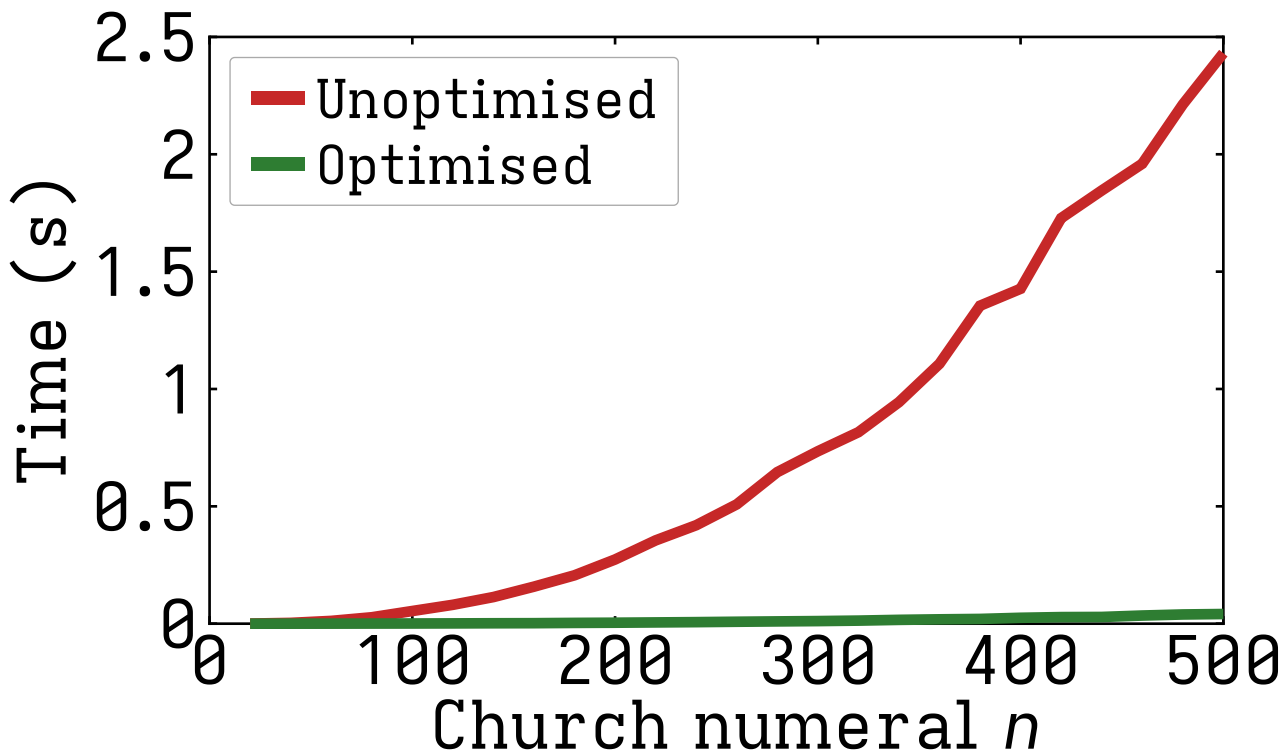
```
@id lastLvl : (n : Nat) → Lvl (1 + n)
@id firstIdx : (n : Nat) → Idx (1 + n)
@id nextIdx : Idx n → Idx (1 + n)

lvlToIdx : (n : Nat) → Lvl n → Idx n
lvlToIdx (1 + n) lz = firstIdx n
lvlToIdx (1 + n) (ls k) = nextIdx (lvlToIdx n k)
lvlToIdx n k @spec n - k - 1 by prf
```

```
map : (@id A → B) → Vect A n → Vect B n
map f [] = []
map f [x, ..xs] = [f x, ..map f xs]
map f @id

weak : Val n → Val (1 + n)
weak (vlam env t) = vlam (map weak env) t
weak (vspine x sp) = vspine (weaken x) (map weak sp)
weak @id
```

NbE performance: normalising Church n^2



We can now customise compilation

- No need to hope for optimisations to fire anymore.
- No need to sacrifice high-level abstractions anymore.
- **Correctness of compilation is preserved.**
  - Can formally prove this as a metatheorem.
  - (as long as the definition of `_ ~ _` is correct)

Now what?

Add more primitives:

- Floats/fixed width ints
- Contiguous arrays
- SIMD, BLAS
- Common rewriting optimisations (e.g. fold fusion)

Example potential projects:

- High-performance safe tensor library
- High-performance compiler with well-formed syntax.

The end