Utrecht University

Programming with dependent types

Beyond evaluating the simply typed lambda calculus

Wouter Swierstra

Utrecht University

From languages to data types

Given a description of a formal language, such as the untyped lambda calculus:

$$t \quad ::= \quad x \quad | \quad t\,t \quad | \quad \lambda x\,t$$

From languages to data types

Given a description of a formal language, such as the untyped lambda calculus:

$$t \quad ::= \quad x \quad | \quad t\,t \quad | \quad \lambda\,x\,t$$

The corresponding *data type* is easy to define:

```
data Term = Var String | App Term Term | Abs String Term
```

From languages to data types

Given a description of a formal language, such as the untyped lambda calculus:

$$t \quad ::= \quad x \;\mid\; t\,t \;\mid\; \lambda x\, t$$

The corresponding *data type* is easy to define:

```
data Term = Var String | App Term Term | Abs String Term
```

We can readily define functions over such data types by induction.

From typed languages to data types...

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

$$\frac{\Gamma \vdash t_1 : \sigma \to \tau \quad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash t_1\ t_2 : \tau}$$

$$\frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x\ t : \sigma \to \tau}$$

From typed languages to data types...

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

$$\frac{\Gamma \vdash t_1 : \sigma \to \tau \quad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash t_1\ t_2 : \tau}$$

$$\frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x\ t : \sigma \to \tau}$$

How do we model such languages?

Representing typed languages

There are several different ways to restrict ourselves to well typed lambda terms:

- define a type checker and (dynamically) assert this invariant;

Representing typed languages

There are several different ways to restrict ourselves to well typed lambda terms:

- define a type checker and (dynamically) assert this invariant;
- using dependent types, introduce a *predicate*, characterizing the untyped lambda terms that may be assigned a valid type. We then only consider the terms that have a proof that they satisfy this predicate;

Representing typed languages

There are several different ways to restrict ourselves to well typed lambda terms:

- define a type checker and (dynamically) assert this invariant;
- using dependent types, introduce a *predicate*, characterizing the untyped lambda terms that may be assigned a valid type. We then only consider the terms that have a proof that they satisfy this predicate;
- introduce a data type *exactly* characterizing the terms of the simply typed lambda calculus.

Towards the simply typed lambda calculus

$$\sigma, \tau \quad ::= \quad \iota \;\mid\; \sigma \to \tau$$

Towards the simply typed lambda calculus

$$\sigma, \tau \quad ::= \quad \iota \quad | \quad \sigma \to \tau$$

Syntax of types:

```
data U : Set where
  ι   : U
  _⇒_  : U → U → U
```

Towards the simply typed lambda calculus

$$\sigma, \tau \quad ::= \quad \iota \ \mid \ \sigma \to \tau$$

Syntax of types:

```
data U : Set where
  ι  : U
  _⇒_  : U → U → U
```

Semantics of types:

```
⟦_⟧ : U → Set
⟦ ι ⟧          = ℕ
⟦ σ ⇒ τ ⟧      = ⟦ σ ⟧ → ⟦ τ ⟧
```

Here we have taken the base type to be natural numbers, but the choice does not matter.

From typing rules to intrinsically typed syntax

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

$$\frac{\Gamma \vdash t_1 : \sigma \to \tau \quad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash t_1\ t_2 : \tau}$$

$$\frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x\ t : \sigma \to \tau}$$

```
data Term : Ctx → U → Set where
  var  : σ ∈ Γ → Term Γ σ
  app  : Term Γ (σ ⇒ t) → Term Γ σ → Term τ
  lam  : Term (σ ∷ Γ) τ → Term Γ (σ ⇒ τ)

Ctx = List U
```

What is still missing?

I haven't defined how to prove σ ∈ Γ – but this is easy less complicated:

```
data _∈_: U → Ctx → Set where
  top  : σ ∈ (σ :: Γ)
  pop  : σ ∈ Γ → σ ∈ (τ :: Γ)
```

If you squint a bit and ignore the fancy types, there are two constructors – just like for Peano naturals.

This representation is sometimes referred to as 'well typed De Bruijn'.

Reflection

Why go through all this trouble?

Reflection

Why go through all this trouble?

Statically enforcing that our terms are well typed:

- guarantee that we do not create unbound variables or ill typed terms;

Reflection

Why go through all this trouble?

Statically enforcing that our terms are well typed:

- guarantee that we do not create unbound variables or ill typed terms;
- makes it *easier* to define functions that *only* work on well typed values;

Reflection

Why go through all this trouble?

Statically enforcing that our terms are well typed:

- guarantee that we do not create unbound variables or ill typed terms;
- makes it *easier* to define functions that *only* work on well typed values;
- most importantly, it is fun and rewarding to program in this style.

Evaluating lambda terms

The evaluator for the simply typed lambda calculus maps terms to their corresponding value:

```
eval : Term Γ σ → ⟦ σ ⟧
eval (app t₁ t₂)  = (eval t₁) (eval t₂)
eval (lam t)      = λ x → eval t
eval (var i)      = ...
```

This almost works...

Environments

To define our evaluator, we need to accumulate an *environment*, storing values for all the free variables:

```
data Env : Ctx → Set where
  nil   : Env []
  cons  : ⟦ σ ⟧ → Env Γ → Env (σ :: Γ)

lookup : Env Γ → σ ∈ Γ → ⟦ u ⟧
```

This is not just a list – but a heterogeneous list, storing values of different types.

Finally, a tagless evaluator

With all these types and definitions, we can finally complete our ‘tagless’ evaluator:

```
eval : Term Γ σ → Env Γ → ⟦ σ ⟧
eval (app t₁ t₂) env  = (eval t₁ env) (eval t₂ env)
eval (lam t) env      = λ x → eval t (cons x env)
eval (var i) env      = lookup env i
```

Finally, a tagless evaluator

With all these types and definitions, we can finally complete our ‘tagless’ evaluator:

```
eval : Term Γ σ → Env Γ → ⟦ σ ⟧
eval (app t₁ t₂) env  = (eval t₁ env) (eval t₂ env)
eval (lam t) env      = λ x → eval t (cons x env)
eval (var i) env      = lookup env i
```

It is amazing that this works at all!

Could you imagine trying to infer the type of `eval`?

Simon Peyton Jones: *show me an example of a program with dependent types…*

Simon Peyton Jones: *show me an example of a program with dependent types…*
…but not the evaluator for the simply typed lambda calculus!

Overview

In the remainder of this talk I want to discuss three such examples, drawn from my own recent work:

- Data structures
- Combinatory logic
- Compiler calculation

All three relate to this ‘simple’ evaluator…

Better data structures

Better data structures

A colleague from the algorithms group might say:

> *Are you fully nerd? You are representing environments as lists. These have linear lookup time and are unsuitable for any term with more than ten variables.*
> *Any reasonable evaluator should have a lookup time logarithmic in the number of bound variables.*

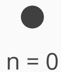
Problem specification

We need to find an alternative data structure:

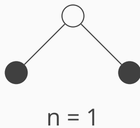
- a ‘flexible array’ supporting `cons` and `lookup` operations;
- all these operations must be total;
- the lookup operation should be logarithmic in the total number of entries;
- finally, the data structure needs to be *heterogeneous*, storing values of different types.

From lists to trees

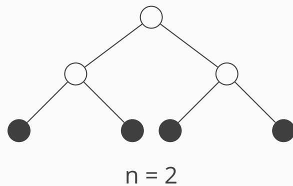
To achieve super linear access, we need to shift from lists to trees.



n = 0



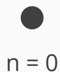
n = 1

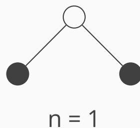


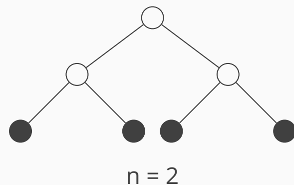
n = 2

From lists to trees

To achieve super linear access, we need to shift from lists to trees.








```
data Tree (a : Set) : ℕ → Set
  leaf  : a → Tree a zero
  node  : Tree a n → Tree a n → Tree a (succ n)
```

Accessing elements in a tree

To denote a particular value stored in a tree of depth n, we need to n steps telling us to continue in the left subtree or the right subtree.

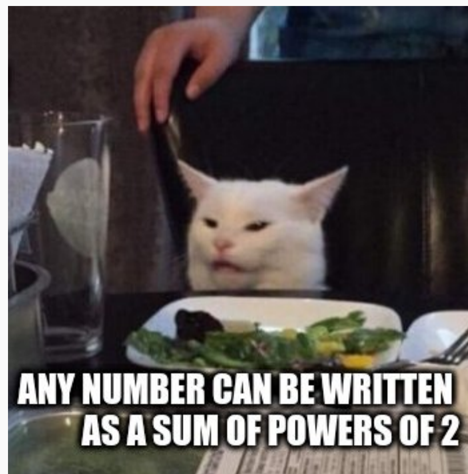
```
data Path : ℕ → Set where
  here   : Path zero
  left   : Path n → Path (succ n)
  right  : Path n → Path (succ n)

lookup : Tree a n → Path n → a
lookup (node t₁ t₂)  (left p)   = lookup t₁ p
lookup (node t₁ t₂)  (right p)  = lookup t₂ p
lookup (leaf x)      here       = x
```

Note: the indices ensure we that this function is total.

YOU CAN'T ASSUME
EVERYTHING IS A POWER OF 2
imgflip.com

YOU CAN'T ASSUME
EVERYTHING IS A POWER OF 2
imgflip.com


ANY NUMBER CAN BE WRITTEN
AS A SUM OF POWERS OF 2

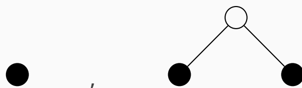
Binary random-access lists

A *binary random-access list* consists of a list of perfect binary trees of increasing depth.

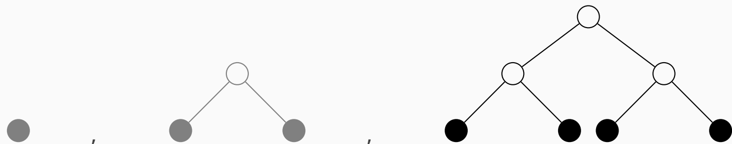
At the i-th position in this list, there may or may not be a perfect binary tree of depth i.

The idea is in Okasaki's classic book on 'Purely functional data structures' – but let's reimplement them enforcing the key invariants using dependent types.

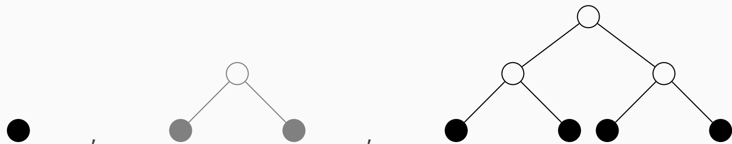
Binary random-access lists storing three elements



Binary random-access lists storing four elements



Binary random-access lists storing five elements



Binary numbers

Every number can be written as a sum of powers of two.

A number's representation in binary determines the shape of the binary random-access list storing that many elements.

Binary numbers

Every number can be written as a sum of powers of two.

A number's representation in binary determines the shape of the binary random-access list storing that many elements.

```
data Bin : Set where
  end   : Bin
  one   : Bin → Bin
  zero  : Bin → Bin

bsucc : Bin → Bin
bsucc end       = one end
bsucc (one b)   = zero (bsucc b)
bsucc (zero b)  = one b
```

Random access lists

```
data RAL (a : Set) (n : Nat) : Bin → Set where
  nil    : RAL a n end
  cons₁  : Tree a n → RAL a (succ n) b → RAL a n (one b)
  cons₀  :            RAL a (succ n) b → RAL a n (zero b)
```

- the binary number counts the number of elements and uniquely determines the shape of our random-access list
- the number n increases as we go down the list – the next tree is going to have more elements (unlike vectors, for example)
- we usually start counting from n = zero, but it's useful to be a bit more general.

Positions and lookup

We can now define a type `Pos n b` that denotes an element stored in a `RAL a n b`:

```
data Pos (n : Nat) : Bin → Set where
  here    : Path n → Pos n (one b)
  there₀  : Pos (succ n) b → Pos n (zero b)
  there₁  : Pos (succ n) b → Pos n (one b)
```

Each position traverses the outer list of trees, ending with a path of depth `n`.

```
lookup : RAL a n b → Pos n b → a
```

Adding elements

Finally, we might want to add new elements to the binary random-access list.

A first attempt might be to define a function such as:

```
add : a → RAL a zero b → RAL a zero (bsucc b)
```

Adding elements

Finally, we might want to add new elements to the binary random-access list.

A first attempt might be to define a function such as:

```
add : a → RAL a zero b → RAL a zero (bsucc b)
```

But we quickly get stuck – we cannot make any recursive calls as the ‘tail’ of the binary random-access list stores larger trees.

Adding elements

Finally, we might want to add new elements to the binary random-access list.

A first attempt might be to define a function such as:

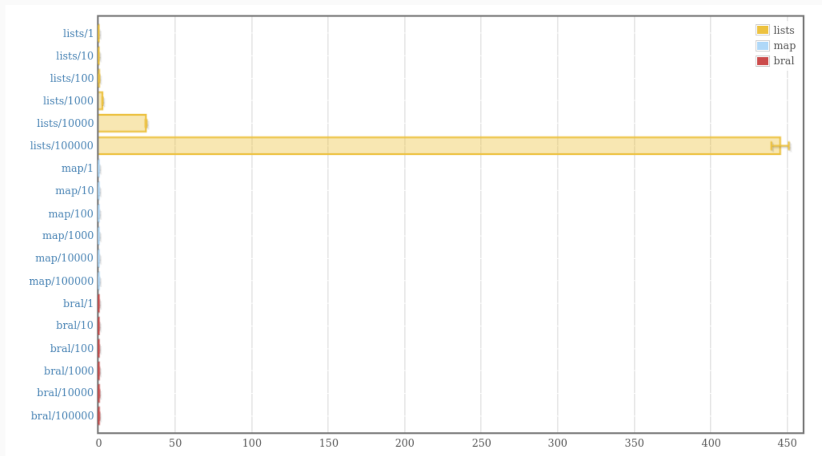
```
add : a → RAL a zero b → RAL a zero (bsucc b)
```

But we quickly get stuck – we cannot make any recursive calls as the 'tail' of the binary random-access list stores larger trees.

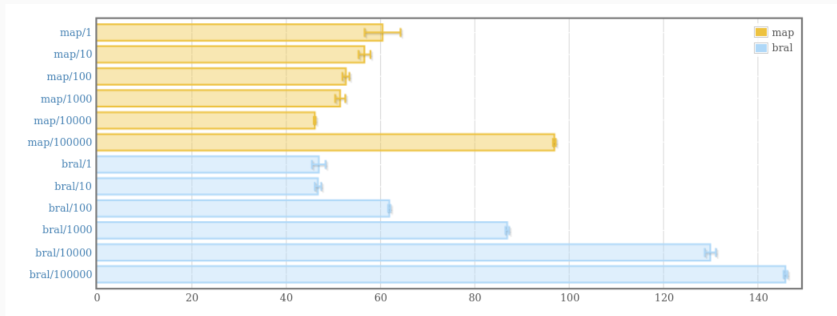
Instead, we need to define a more general operation that adds a tree of depth `n` to a binary random-access list:

```
addTree : Tree a n → RAL a n b → RAL a n (bsucc b)
```

Benchmarking in Haskell (homogeneous)



Benchmarking in Haskell (homogeneous)



Heterogeneous binary random access lists

- We can extend this to the heterogeneous case:

```
data HRAL : RAL U n b → Set where ...
```

- Despite the apparent complexity, writing an 'efficient' lambda calculus evaluator written using heterogeneous binary random-access lists is no harder than using heterogeneous lists.
- The only hard part is converting 'linear' positions, $\sigma \in \Gamma$, to their corresponding position in a random access list.
- Almost no code needs to change, compared to our previous implementation.

Conversion to combinatory logic

Conversion to combinatory logic

A colleague from logic department might say:

> *Are you fully nerd? The lambda calculus is a language with binding! Working with first order term rewriting systems, such as those given by combinatory logic, is much easier.*
> *Can't you convert your well typed lambda terms to combinatory logic? And prove that the translation preserves types and semantics?*

Combinatory logic

A language without lambdas:

$$t \quad ::= \quad x \quad | \quad t\ t \quad | \quad \mathsf{S} \mid \quad \mathsf{K} \mid \quad \mathsf{I}$$

Instead of beta reduction or substitution, we have three simple reduction rules:

$$\mathsf{I}\ x \longrightarrow x$$

$$\mathsf{K}\ x\ y \longrightarrow x$$

$$\mathsf{S}\ x\ y\ z \longrightarrow (x\ z)\ (y\ z)$$

Combinatory logic

A language without lambdas:

$$t \quad ::= \quad x \quad | \quad t\ t \quad | \quad \mathsf{S} \mid \mathsf{K} \mid \mathsf{I}$$

Instead of beta reduction or substitution, we have three simple reduction rules:

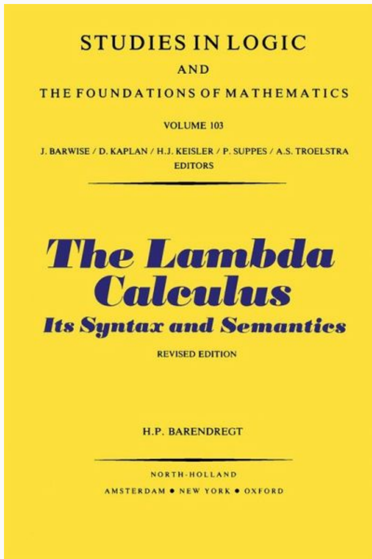
$$\mathsf{I}\ x \longrightarrow x$$

$$\mathsf{K}\ x\ y \longrightarrow x$$

$$\mathsf{S}\ x\ y\ z \longrightarrow (x\ z)\ (y\ z)$$

Somewhat surprisingly, these three combinators suffice to mimic lambda abstraction!

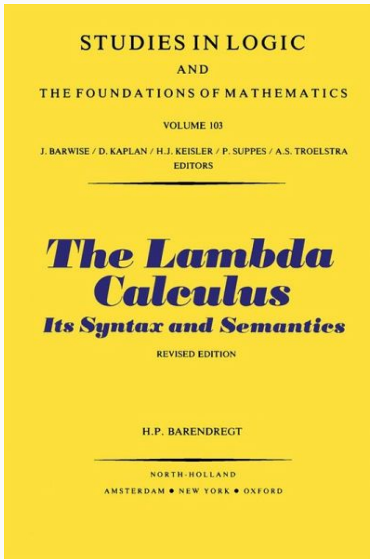
Bracket abstraction



The translation from (untyped) lambda terms to combinatory logic is well established (Chapter 7).

We map variables (in lambda calculus) to variables (in combinatory logic); we map applications to applications.

What about lambda abstractions?



To translate lambda abstractions to combinatory logic, we need an auxiliary function, sometimes referred to as *bracket abstraction*.

The key idea is that, after translating the body of a lambda, we add additional combinators to ensure the (bound) variable is replaced accordingly.

Datatype design

$$t \quad ::= \quad x \quad | \quad t\,t \quad | \quad \mathsf{S} \;|\; \mathsf{K} \;|\; \mathsf{I}$$

What should the type of our combinatory terms be?

```
data CL : Set
  var : String → CL
  app : CL → CL → CL
  S K I : CL
```

Datatype design

$$t \quad ::= \quad x \quad | \quad t\,t \quad | \quad \mathsf{S} \,|\, \mathsf{K} \,|\, \mathsf{I}$$

What should the type of our combinatory terms be?

```
data CL : Set
  var : String → CL
  app : CL → CL → CL
  S K I : CL
```

This representation is not wrong – but it doesn't tell us anything about the types involved!

```
translate : Term Γ σ → CL
```

In particular, there is no guarantee that this function maps well typed terms to well typed combinators.

Type preserving translation

To ensure types are preserved, we instead define a datatype for well typed terms in combinatory logic:

```
data CL (Γ : Ctx) : U → Set where
  var : σ ∈ Γ → CL Γ σ
  app : CL Γ (σ ⇒ τ) → CL Γ σ → CL Γ τ
  I   : CL Γ (σ ⇒ σ)
  K   : CL Γ (σ ⇒ τ ⇒ σ)
  S   : ...
```

Type preserving translation

To ensure types are preserved, we instead define a datatype for well typed terms in combinatory logic:

```
data CL (Γ : Ctx) : U → Set where
  var : σ ∈ Γ → CL Γ σ
  app : CL Γ (σ ⇒ τ) → CL Γ σ → CL Γ τ
  I   : CL Γ (σ ⇒ σ)
  K   : CL Γ (σ ⇒ τ ⇒ σ)
  S   : ...
```

Now it is clear that the translation preserves types:

```
translate : Term Γ σ → CL Γ σ
```

Translation to combinatory logic

Most of the translation is now trivial:

```
translate : Term Γ σ → CL Γ σ
translate (app t₁ t₂) = app (translate t₁) (translate t₂)
translate (var i)     = var i
translate (lam t)     = abs (translate t)
  where
  abs : CL (σ :: Γ) τ → CL Γ (σ ⇒ τ)
```

The only real work is done by the bracket abstraction function, `abs`.

Bracket abstraction - the theory

... is not closed under the rule ξ, i.e. in general ... example $\lambda \vdash \mathbf{SK} = \mathbf{KI}$, but $CL \nvdash SK$ ...

$$CL \vdash P = Q \not\Rightarrow CL \vdash \lambda^* x.P = \lambda^* x.Q.$$

For example $CL \vdash Ix = x$, but $CL \nvdash S(KI)I = I$.

Curry extended CL by a finite set A_β of closed equations such that $CL + A_\beta$ is equivalent to $\boldsymbol{\lambda}$. We follow his construction.

First an auxiliary abstraction operator is introduced.

7.3.4. Definition. $\lambda_1 x.P$ is defined by induction on the structure of P.

$$\lambda_1 x.x \equiv I,$$

$$\lambda_1 x.c \equiv Kc \quad \text{if } c \text{ is a variable} \not\equiv x \text{ or } c \in \{K, S\},$$

$$\lambda_1 x.PQ \equiv S(\lambda_1 x.P)(\lambda_1 x.Q).$$

Let A be a set of equations between closed CL-terms. $CL + A$ is the theory CL extended by the elements of A as axioms.

7.3.5. Lemma. *Consider the schemes*

Bracket abstraction – the implementation

```
-- the 'same' type as the lam constructor
abs : CL (σ ∷ Γ) τ → CL Γ (σ ⇒ τ)
-- the most recently bound variable is replaced by I
abs (var top)     = I
-- other variables/combinators are wrapped by K
abs (var (pop j)) = app K (var j)
abs S             = app K S
abs K             = app K K
abs I             = app K I
-- application introduces S and recurses
abs (app t₁ t₂)   = app (app S (abs t₁)) (abs t₂)
```

All the types go through easily enough...

Bracket abstraction – the implementation

```
-- the 'same' type as the lam constructor
abs : CL (σ ∷ Γ) τ → CL Γ (σ ⇒ τ)
-- the most recently bound variable is replaced by I
abs (var top)     = I
-- other variables/combinators are wrapped by K
abs (var (pop j)) = app K (var j)
abs S             = app K S
abs K             = app K K
abs I             = app K I
-- application introduces S and recurses
abs (app t₁ t₂)   = app (app S (abs t₁)) (abs t₂)
```

All the types go through easily enough…

What about proving correctness?

Correctness – take 1

We define an evaluator for combinatory terms:

```
evalCL : CL Γ σ → Env Γ → ⟦ σ ⟧
evalCL K env         = λ x y → x
evalCL I env         = λ x → x
evalCL (app f e) env = (evalCL f env) (evalCL e env)
...
```

Now we only need to prove the obvious correctness statement:

```
correctness : (t : Term Γ σ) (env : Env Γ) → evalCL (translate t) env ≡ eval t env
```

Correctness – take 1

The proof itself is not very interesting – it requires one additional lemma:

```
abs-correct : (t : CL (σ ∷ Γ) τ) (env : Env Γ) (v : ⟦ σ ⟧) →
  evalCL (abs t) env v ≡ evalCL t (v ∷ env)
```

This lemma makes precise that ‘bracket abstraction behaves like lambda abstraction’.

The proof follows from immediate induction.

Correctness – take 1

The proof itself is not very interesting – it requires one additional lemma:

```
abs-correct : (t : CL (σ :: Γ) τ) (env : Env Γ) (v : ⟦ σ ⟧) →
  evalCL (abs t) env v ≡ evalCL t (v :: env)
```

This lemma makes precise that ‘bracket abstraction behaves like lambda abstraction’.

The proof follows from immediate induction.

If the proof is so obvious – can we make our translation correct by construction?

Correctness - take 2

Let's redefine our datatype for terms in combinatory logic.

Now they not only carry their *type* but also their (dynamic) semantics, i.e., the result of evaluation:

```
data CL : (Γ : Ctx) → (σ : U) → (Env Γ → ⟦ σ ⟧) → Set where
  K    : CL Γ (σ ⇒ (τ ⇒ σ)) (λ env → λ x y → x)
  I    : CL Γ (σ ⇒ σ) (λ env → λ x → x)
  app  : CL Γ (σ ⇒ τ) f → CL Γ σ t → CL Γ τ (f t)
  ...
```

Correctness - take 2

Let's redefine our datatype for terms in combinatory logic.

Now they not only carry their *type* but also their (dynamic) semantics, i.e., the result of evaluation:

```
data CL : (Γ : Ctx) → (σ : U) → (Env Γ → ⟦ σ ⟧) → Set where
  K    : CL Γ (σ ⇒ (τ ⇒ σ)) (λ env → λ x y → x)
  I    : CL Γ (σ ⇒ σ) (λ env → λ x → x)
  app  : CL Γ (σ ⇒ τ) f → CL Γ σ t → CL Γ τ (f t)
  ...
```

Now we can *specify* the translation to combinatory logic that is correct by construction:

```
translate : (t : Term Γ σ) → CL Γ σ (eval t)
```

Correctness - take 2

Let's redefine our datatype for terms in combinatory logic.

Now they not only carry their *type* but also their (dynamic) semantics, i.e., the result of evaluation:

```
data CL : (Γ : Ctx) → (σ : U) → (Env Γ → ⟦ σ ⟧) → Set where
  K    : CL Γ (σ ⇒ (τ ⇒ σ)) (λ env → λ x y → x)
  I    : CL Γ (σ ⇒ σ) (λ env → λ x → x)
  app  : CL Γ (σ ⇒ τ) f → CL Γ σ t → CL Γ τ (f t)
  ...
```

Now we can *specify* the translation to combinatory logic that is correct by construction:

```
translate : (t : Term Γ σ) → CL Γ σ (eval t)
```

How is this function defined?

Translation, revisited

The translation remains exactly the same – it is only the types that change!

```
translate : (t : Term Γ σ) → CL Γ σ (eval t)
translate (app t₁ t₂) = app (translate t₁) (translate t₂)
translate (var i)     = var i
translate (lam t)     = abs (translate t)
```

Translation, revisited

The translation remains exactly the same – it is only the types that change!

```
translate : (t : Term Γ σ) → CL Γ σ (eval t)
translate (app t₁ t₂) = app (translate t₁) (translate t₂)
translate (var i)     = var i
translate (lam t)     = abs (translate t)
```

Of course, we still need to (re)define the bracket abstraction function…

Bracket abstraction - correct by construction

```
abs : ∀ {f} → CL (σ :: Γ) τ f → CL Γ (σ ⇒ τ) (λ env x → f (x :: env))
```

Do you recognize the right hand side of our evaluator in the *type* of `abs`?

Bracket abstraction - correct by construction

```
abs : ∀ {f} → CL (σ ∷ Γ) τ f → CL Γ (σ ⇒ τ) (λ env x → f (x ∷ env))
```

Do you recognize the right hand side of our evaluator in the *type* of `abs`?

But once again, the *definition* remains unchanged; it is only the types that are richer.

Review

This defines a correct by construction conversion to combinators:

```
translate : (t : Term Γ σ) → CL Γ σ (eval t)
```

The function's type specifies that it preserves types and semantics.

Review

This defines a correct by construction conversion to combinators:

```
translate : (t : Term Γ σ) → CL Γ σ (eval t)
```

The function's type specifies that it preserves types and semantics.

And we didn't have to do any proofs to establish this!

Review

This defines a correct by construction conversion to combinators:

```
translate : (t : Term Γ σ) → CL Γ σ (eval t)
```

The function's type specifies that it preserves types and semantics.

And we didn't have to do any proofs to establish this!

We even extend the translation to use additional combinators, B and C, to reduce the number of reduction steps required.

Review

This defines a correct by construction conversion to combinators:

```
translate : (t : Term Γ σ) → CL Γ σ (eval t)
```

The function's type specifies that it preserves types and semantics.

And we didn't have to do any proofs to establish this!

We even extend the translation to use additional combinators, B and C, to reduce the number of reduction steps required.

But why does this work at all?

Correct by construction

This seems like a ‘parlour trick’ – aren’t I clever!

Correct by construction

This seems like a ‘parlour trick’ – aren’t I clever!

But the meta-properties that makes this possible are:

- the translation is defined by simple induction, with one auxiliary function;
- the correctness proof proceeds by immediate induction, using one additional lemma about this auxiliary function;
- the structure of the translation and correctness proof coincide *exactly*;

So why not do both at once?

Calculating compilers

A colleague from engineering department might say:

> *By itself, the lambda calculus is not that interesting. I want to have a compiler from the lambda calculus to some (abstract) machine!*
> *Can you not derive a correct compiler from its evaluator?*

Existing work...

2.2 *Step 2 – Transform into a stack transformer*

The next step is to transform the evaluation function into a version that utilises a stack, in order to make the manipulation of argument values explicit. In particular, rather than returning a single value of type *Int*, we seek to derive a more general evaluation function, $eval_S$, that takes a stack of integers as an additional argument, and returns a modified stack given by pushing the value of the expression onto the top of the stack. More precisely, if we represent a stack as a list of integers (where the head is the top element)

$$\textbf{type}\ Stack = [Int]$$

Calculating correct compilers, Bahr and Hutton (2015)

Existing work...

2.2 Compiler Specification

We now seek to calculate a compiler for our well-typed expression language, using the approach developed by Bahr and Hutton [2015]. In that setting, the target language for the compiler is a stack-based virtual machine in which a stack is represented simply as a list of numbers. In our setting, we can take advantage of the power of dependent types and use a more refined stack type that is indexed by the types of the elements it contains [Poulsen et al. 2018]:

```
variable
  S S' S'' : List Set

data Stack : List Set → Set where
  ϵ : Stack []
  _▷_ : T → Stack S → Stack (T :: S)
```

Calculating Dependently-Typed Compilers, Pickard and Hutton (2021)

Existing work...

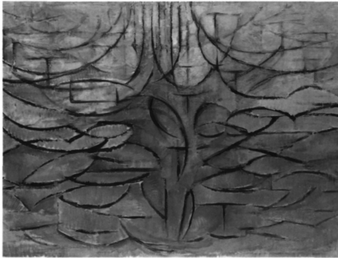
2.3 Compiler Specification

We have now defined a type *Expr* that represents the syntax of a simple expression language that supports printing, and a function *eval* that gives a monadic evaluation semantics for the language in terms of print sequences. In this section we show how to specify the desired behaviour of a compiler for this simple expression language.

Our goal is to define a function $comp :: Expr \rightarrow Code$ that compiles an expression into code for a suitable machine. We assume the compiler targets a stack-based machine, whose semantics is given a function $exec :: Code \rightarrow Stack \rightarrow Stack$ that executes code using an initial stack to give a final stack, where a stack is simply a list of integers:

type $Stack = [Int]$

Calculating compilers effectively, Garby, Hutton & Bahr (2024)



Erik Meijer's PhD thesis (1992)

Explores, among other things, how to shift between stack-based and direct evaluation

More than one semantics...

Consider 'the essence of stack computation' (Eliott 2020):

$$a \to b \quad \simeq \quad \forall c \,.\, a \times c \to b \times c$$

Our previous semantics for types

```
⟦_⟧ : U → Set
⟦ ι ⟧         = ℕ
⟦ σ ⇒ τ ⟧     = ⟦ σ ⟧ → ⟦ τ ⟧
```

Can we define an alternative interpretation corresponding to the right hand side of this isomorphism?

More than one semantics...

```
⟦_⟧ₛ    : U → Set
⟦ σ ⟧ₛ  = ∀ ξ → args σ ξ → result σ ξ
```

Where args and res collect the arguments and result type respectively:

```
args : U → Set → Set
args (σ ⇒ τ) ξ  = ⟦ σ ⟧ₛ × args τ ξ
args ι       ξ  = ξ

res : U → Set → Set
res (σ ⇒ τ) ξ  = res τ ξ
res ι       ξ  = ⟦ ι ⟧ × ξ
```

Example: addition

Many compiler calculation papers start with a language for simple arithmetic expressions:

$$e \quad ::= \quad n \quad | \quad e + e$$

Example: addition

Many compiler calculation papers start with a language for simple arithmetic expressions:

$$e \quad ::= \quad n \quad | \quad e + e$$

We now have *two* different types to associate with an operator for addition, `ι ⇒ ι ⇒ ι`,

- the 'regular' semantics: `ℕ → ℕ → ℕ`
- the 'stack based' semantics: `∀ ξ → (ℕ × ℕ × ξ) → (ℕ × ξ)`

The *type* of our stacks and stack-based operations are all *computed* directly.

Conversions

Not only can we define a different *interpretation* for our types, we define generic *conversions* between the two:

```
γ : ∀ σ → ⟦ σ ⟧ₛ → ⟦ σ ⟧
α : ∀ σ → ⟦ σ ⟧  → ⟦ σ ⟧ₛ
```

The two functions are defined by induction on the *type* σ, shuffling arguments to and from the stack – essentially witnessing a generalized uncurry-curry isomorphism.

Conversions

Not only can we define a different *interpretation* for our types, we define generic *conversions* between the two:

$$\gamma : \forall\ \sigma \to [\![\ \sigma\ ]\!]_s \to [\![\ \sigma\ ]\!]$$
$$\alpha : \forall\ \sigma \to [\![\ \sigma\ ]\!] \to [\![\ \sigma\ ]\!]_s$$

The two functions are defined by induction on the *type* σ, shuffling arguments to and from the stack – essentially witnessing a generalized uncurry-curry isomorphism.

We show (under suitable assumptions) that these two are mutual inverses.

Beyond arithmetic...

- The same stack-based semantics and isomorphism works for other languages...

Beyond arithmetic…

- The same stack-based semantics and isomorphism works for other languages…
- Including the simply typed lambda calculus!

Beyond arithmetic…

- The same stack-based semantics and isomorphism works for other languages…
- Including the simply typed lambda calculus!
- Meijer’s thesis explores this idea, including several simple optimizations.

Beyond arithmetic…

- The same stack-based semantics and isomorphism works for other languages…
- Including the simply typed lambda calculus!
- Meijer's thesis explores this idea, including several simple optimizations.
- Each of which follows naturally in this setting too.

So what?

This lets us derive a compiler in several steps:

- Start with a intrinsically typed evaluator, like the one for the simply typed lambda calculus;
- Convert the evaluator to its stack-based equivalent;
- CPS transforming the stack based evaluator fixes evaluation order;
- Defunctionalising yields an abstract machine (à la Danvy);
- Reforestation yields (instructions for) a compiler (à la Hutton).

The second step *requires* a shift in types – we need to compute new types – a third example of dependent types!

Summary

This gives three separate examples, all revolving around the evaluator for the simply typed lambda calculus:

- using dependent types to enforce data structure *invariants*;
- using dependent types to develop a *correct by construction* translation function;
- using dependent types to *compute* new types.

Bibliography

1. Heterogeneous binary random-access lists; JFP, 2020, Vol. 30.
2. A correct-by-construction conversion to combinators; JFP, vol 33, 2023.
3. A type directed calculation of an intrinsically typed optimizing compiler; under review for JFP.

Questions?