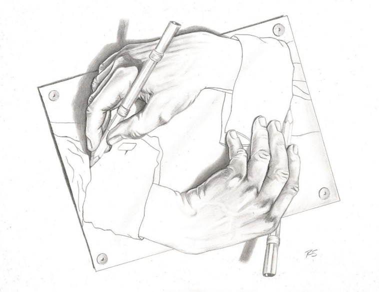
Introduction to Theoretical Computer Science

Lecture 15: Recursion and Typed λ-Calculus

Dr. Liam O'Connor

University of Edinburgh
Semester 1, 2023/2024

Puzzle



Puzzle

Find a λ-term $\mathcal{Y}$ such that

$$\mathcal{Y}\, f \quad \mapsto_\beta^\star \quad f\ (\mathcal{Y}\, f)$$

Can you use this to define recursive functions? e.g. factorial?

The Y Combinator

The term we're looking for is called a fixed point combinator. And they're the way we achieve recursion in the λ-calculus.

Example (Recursive functions)

Exercise: Assuming a definition for $\mathcal{Y}$, as well as If, Equal, Add and Suc, define a recursive function to compute the sum of every natural number from a given a to a given b.

The Y Combinator

The term we're looking for is called a fixed point combinator. And they're the way we achieve recursion in the λ-calculus.

Example (Recursive functions)

Exercise: Assuming a definition for $\mathcal{Y}$, as well as If, Equal, Add and Suc, define a recursive function to compute the sum of every natural number from a given a to a given b.

0 To find $\mathcal{Y}$, we can draw inspiration from our previous diverging example:

$$(\lambda x.\ (x\ x))\ (\lambda x.\ (x\ x))$$

The Y Combinator

The term we're looking for is called a fixed point combinator. And they're the way we achieve recursion in the λ-calculus.

Example (Recursive functions)

Exercise: Assuming a definition for $\mathcal{Y}$, as well as If, Equal, Add and Suc, define a recursive function to compute the sum of every natural number from a given a to a given b.

0 To find $\mathcal{Y}$, we can draw inspiration from our previous diverging example:

$$(\lambda x.\ (x\ x))\ (\lambda x.\ (x\ x))$$

and define $\mathcal{Y}$ as follows:

$$\mathcal{Y} \equiv (\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))$$

Exercise: Let's demonstrate that $\mathcal{Y}\ g \equiv_\beta g\ (\mathcal{Y}\ g)$

Higher Order Logic

Originally, λ-calculus was intended for use as a term language for a logic, called *higher-order logic*. The existence of terms like $\mathcal{Y}$ poses a problem for this, as, for example:

$$\mathcal{Y}\,\neg \equiv_{\beta} \neg\,(\mathcal{Y}\,\neg)$$

It certainly isn't good to have a logical term that is equal to its own negation! Church solves this with *types*.

Higher Order Logic

Originally, λ-calculus was intended for use as a term language for a logic, called *higher-order logic*. The existence of terms like $\mathcal{Y}$ poses a problem for this, as, for example:

$$\mathcal{Y}\,\neg \equiv_\beta \neg\,(\mathcal{Y}\,\neg)$$

It certainly isn't good to have a logical term that is equal to its own negation! Church solves this with *types*.

Adding Types

- Fix a set of *base types* ($\texttt{nat}, \texttt{bool}$, etc.)

Higher Order Logic

Originally, λ-calculus was intended for use as a term language for a logic, called *higher-order logic*. The existence of terms like $\mathcal{Y}$ poses a problem for this, as, for example:

$$\mathcal{Y}\,\neg \equiv_\beta \neg\,(\mathcal{Y}\,\neg)$$

It certainly isn't good to have a logical term that is equal to its own negation! Church solves this with *types*.

Adding Types

- Fix a set of *base types* ($\mathtt{nat}$, $\mathtt{bool}$, etc.)
- If σ and τ are types, then $\sigma \to \tau$ is a type of a *function* from σ to τ. Like Haskell, it is right-associative: $\sigma \to \tau \to \rho = \sigma \to (\tau \to \rho)$

Higher Order Logic

Originally, λ-calculus was intended for use as a term language for a logic, called *higher-order logic*. The existence of terms like $\mathcal{Y}$ poses a problem for this, as, for example:

$$\mathcal{Y}\ \neg \equiv_\beta \neg\ (\mathcal{Y}\ \neg)$$

It certainly isn't good to have a logical term that is equal to its own negation! Church solves this with *types*.

Adding Types

- Fix a set of *base types* ($\mathtt{nat}, \mathtt{bool}$, etc.)
- If σ and τ are types, then $\sigma \to \tau$ is a type of a *function* from σ to τ. Like Haskell, it is right-associative: $\sigma \to \tau \to \rho = \sigma \to (\tau \to \rho)$
- A λ-abstraction now additionally specifies the type of the parameter: $\lambda x : \tau.\ t$

Natural Deduction

Logic and Types

We can specify a logical system as a *deductive system* by providing a set of rules and axioms that describe how to prove various connectives. We can specify typing the same way!

For example, to prove a λ-abstraction $\lambda x : \sigma.\ t$ has type $\sigma \to \tau$, we must show that the function body t has type τ assuming x has type σ. This rule is written as:

$$\frac{x : \sigma, \Gamma \vdash t : \tau}{\Gamma \vdash (\lambda x : \sigma.\ t) : \sigma \to \tau} \to_I$$

derivability

(if the top, then the bottom)

entailment

(assuming the left, we can prove the right)

Typing

The full set of rules for the *simply typed λ-calculus* is as follows:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}A \qquad \frac{x : \sigma, \Gamma \vdash t : \tau}{\Gamma \vdash (\lambda x : \sigma.\ t) : \sigma \to \tau}\to_I$$

$$\frac{\Gamma \vdash t : \sigma \to \tau \qquad \Gamma \vdash u : \sigma}{\Gamma \vdash t\ u : \tau}\to_E$$

Example (Typing)

- By drawing a *proof tree*, and assuming Add has type $\mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}$, show that $(\lambda x : \mathtt{nat}.\ \mathsf{Add}\ x\ x)$ has type $\mathtt{nat} \to \mathtt{nat}$
- Show that our non-terminating term $(\lambda x.\ x\ x)(\lambda x.\ x\ x)$ cannot be typed. Similarly show that $\mathcal{Y}$ cannot be typed.

Some Results

Uniqueness of types In a given context (types for free variables), any simply typed λ-terms has at most one type. Deciding this is in **P**.

Some Results

Uniqueness of types In a given context (types for free variables), any simply typed λ-terms has at most one type. Deciding this is in **P**.

Subject reduction (type safety) Typing respects $\equiv_{\alpha\beta\eta}$, i.e. reduction does not affect a term's type.

Some Results

Uniqueness of types In a given context (types for free variables), any simply typed λ-terms has at most one type. Deciding this is in **P**.

Subject reduction (type safety) Typing respects $\equiv_{\alpha\beta\eta}$, i.e. reduction does not affect a term's type.

Strong normalisation Any well-typed term evaluates in finitely many reductions to a unique irreducible term. If the type is a base type, this term is a constant.

Some Results

Uniqueness of types In a given context (types for free variables), any simply typed λ-terms has at most one type. Deciding this is in **P**.

Subject reduction (type safety) Typing respects $\equiv_{\alpha\beta\eta}$, i.e. reduction does not affect a term's type.

Strong normalisation Any well-typed term evaluates in finitely many reductions to a unique irreducible term. If the type is a base type, this term is a constant.

We lost recursion!

We have seen that $\mathcal{Y}$ cannot be typed, and strong normalisation means that no such combinator could exist in simply typed λ-calculus.

Adding recursion back in

If we want to do general computation in our λ-calculus, we need recursion back. So, we just extend the typed λ-calculus with a new built-in feature, called **fix**:

$$\frac{\Gamma \vdash t : \tau \to \tau}{\Gamma \vdash \textbf{fix}\ t : \tau}$$

And we extend β-reduction to unroll our recursion one step:

$$\textbf{fix}\ (\lambda x : \tau.\ t) \quad \mapsto_\beta \quad t[{}^{\textbf{fix}\ (\lambda x:\tau.\ t)}/_{x}]$$

Now we can use **fix** as we used $\mathcal{Y}$ in our untyped setting.

Adding recursion back in

If we want to do general computation in our λ-calculus, we need recursion back. So, we just extend the typed λ-calculus with a new built-in feature, called **fix**:

$$\frac{\Gamma \vdash t : \tau \to \tau}{\Gamma \vdash \textbf{fix}\ t : \tau}$$

And we extend β-reduction to unroll our recursion one step:

$$\textbf{fix}\ (\lambda x : \tau.\ t) \quad \mapsto_\beta \quad t[{}^{\textbf{fix}\ (\lambda x:\tau.\ t)}/_x]$$

Now we can use **fix** as we used $\mathcal{Y}$ in our untyped setting.

Total Programming

Some type-theoretic languages (Agda, Idris) avoid adding general recursion to their underlying λ-calculus. Let's talk about why they did that!

Product Types

Lets extend our simple lambda calculus with some other composite types, such as *product types* or tuples:

$$\tau_1 \times \tau_2$$

We won't have type declarations, named fields or anything like that. More than two values can be combined by nesting products, for example a three dimensional vector:

$$\mathtt{nat} \times (\mathtt{nat} \times \mathtt{nat})$$

Constructors and Eliminators

We can construct a product type similarly to Haskell tuples:

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \times_I$$

The only way to extract each component of the product is to use the `fst` and `snd` eliminators:

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathrm{fst}\ e : \tau_1} \times_{E1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathrm{snd}\ e : \tau_2} \times_{E2}$$

Semantics

We extend our notion of β-reduction to describe how these new built-in features evaluate:

$$\mathsf{fst}\ (v_1, v_2) \mapsto_\beta v_1 \qquad \mathsf{snd}\ (v_1, v_2) \mapsto_\beta v_2$$

Unit Types

Currently, we have no way to express a type with just one value. This may seem useless at first, but it becomes useful in combination with other types.
We'll introduce a new base type, $\mathbf{1}$, pronounced *unit*, that has exactly one inhabitant, written $()$:

$$\frac{}{\Gamma \vdash () : \mathbf{1}}\ \mathbf{1}_I$$

Disjunctive Composition

We can't, with just our product types, express a type with exactly three values.

Example (Trivalued type)

```
data TrafficLight = Red | Amber | Green
```

Disjunctive Composition

We can't, with just our product types, express a type with exactly three values.

Example (Trivalued type)

```
data TrafficLight = Red | Amber | Green
```

In general we want to express data that can be one of multiple alternatives, that contain different bits of data.

Example (More elaborate alternatives)

```
type Length = Int
type Angle = Int
data Shape = Rect Length Length
           | Circle Length | Point
           | Triangle Angle Length Length
```

Sum Types

We will use *sum types* to express the possibility that data may be one of two forms.

$$\tau_1 + \tau_2$$

This is similar to the Haskell `Either` type.
Our `TrafficLight` type can be expressed (grotesquely) as a sum of units:

$$\texttt{TrafficLight} \simeq \mathbf{1} + (\mathbf{1} + \mathbf{1})$$

Constructors and Eliminators for Sums

To make a value of type $\tau_1 + \tau_2$, we invoke one of two constructors:

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{InL}\ e : \tau_1 + \tau_2} +_{I1} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{InR}\ e : \tau_1 + \tau_2} +_{I2}$$

We can branch based on which alternative is used using pattern matching:

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad x : \tau_1, \Gamma \vdash e_1 : \tau \qquad y : \tau_2, \Gamma \vdash e_2 : \tau}{\Gamma \vdash (\mathbf{case}\ e\ \mathbf{of}\ \mathsf{InL}\ x \to e_1; \mathsf{InR}\ y \to e_2) : \tau} +_E$$

Examples

Example (Traffic Lights)

Our traffic light type has three values as required:

$$\begin{aligned}
\texttt{TrafficLight} &\simeq \mathbf{1} + (\mathbf{1} + \mathbf{1}) \\
\\
\texttt{Red} &\simeq \mathsf{InL}\ () \\
\texttt{Amber} &\simeq \mathsf{InR}\ (\mathsf{InL}\ ()) \\
\texttt{Green} &\simeq \mathsf{InR}\ (\mathsf{InR}\ ())
\end{aligned}$$

Semantics

$$(\textbf{case}\ (\mathsf{InL}\ v)\ \textbf{of}\ \mathsf{InL}\ x \rightarrow\ e_1; \mathsf{InR}\ y \rightarrow\ e_2) \mapsto_\beta e_1[^v/_x]$$
$$(\textbf{case}\ (\mathsf{InR}\ v)\ \textbf{of}\ \mathsf{InL}\ x \rightarrow\ e_1; \mathsf{InR}\ y \rightarrow\ e_2) \mapsto_\beta e_2[^v/_y]$$

Semantics

$$(\textbf{case}\ (\mathsf{InL}\ v)\ \textbf{of}\ \mathsf{InL}\ x \rightarrow\ e_1;\ \mathsf{InR}\ y \rightarrow\ e_2) \mapsto_\beta e_1[{}^{v}/_{x}]$$

$$(\textbf{case}\ (\mathsf{InR}\ v)\ \textbf{of}\ \mathsf{InL}\ x \rightarrow\ e_1;\ \mathsf{InR}\ y \rightarrow\ e_2) \mapsto_\beta e_2[{}^{v}/_{y}]$$

The Empty Type

We add another type, called $\mathbf{0}$, that has no inhabitants.
Because it is empty, there is no way to construct it.
We do have a way to eliminate it, however:

$$\frac{\Gamma \vdash e : \mathbf{0}}{\Gamma \vdash \mathsf{absurd}\ e : ?}\,\mathbf{0}_E$$

The Empty Type

We add another type, called $\mathbf{0}$, that has no inhabitants.
Because it is empty, there is no way to construct it.
We do have a way to eliminate it, however:

$$\frac{\Gamma \vdash e : \mathbf{0}}{\Gamma \vdash \mathsf{absurd}\ e : \tau}\,\mathbf{0}_E$$

If I have a variable of the empty type in scope, we must be looking at an expression that will never be evaluated.
Therefore, we can assign any type we like to this expression, because it will never be executed.

Examining our Types

Lets look at the rules for typed lambda calculus extended with sums and products:

$$\frac{\Gamma \vdash e : \mathbf{0}}{\Gamma \vdash \mathsf{absurd}\ e : \tau} \qquad \frac{}{\Gamma \vdash () : \mathbf{1}}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{InL}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{InR}\ e : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad x : \tau_1, \Gamma \vdash e_1 : \tau \qquad y : \tau_2, \Gamma \vdash e_2 : \tau}{\Gamma \vdash (\mathbf{case}\ e\ \mathbf{of}\ \mathsf{InL}\ x \to e_1; \mathsf{InR}\ y \to e_2) : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{fst}\ e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{snd}\ e : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad \frac{x : \tau_1, \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2}$$

Squinting a Little

Lets remove all the terms, leaving just the types and the contexts:

$$\frac{\Gamma \vdash \mathbf{0}}{\Gamma \vdash \tau} \qquad \frac{}{\Gamma \vdash \mathbf{1}}$$

$$\frac{\Gamma \vdash \tau_1}{\Gamma \vdash \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash \tau_1 + \tau_2 \qquad \tau_1, \Gamma \vdash \tau \qquad \tau_2, \Gamma \vdash \tau}{\Gamma \vdash \tau}$$

$$\frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash \tau_1 \times \tau_2}{\Gamma \vdash \tau_1} \qquad \frac{\Gamma \vdash \tau_1 \times \tau_2}{\Gamma \vdash \tau_2}$$

$$\frac{\Gamma \vdash \tau_1 \to \tau_2 \qquad \Gamma \vdash \tau_1}{\Gamma \vdash \tau_2} \qquad \frac{\tau_1, \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \to \tau_2}$$

Does this resemble anything you've seen before?

A surprising coincidence!

Types are exactly the same structure as *intuitionistic logic*:

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash P} \qquad \frac{}{\Gamma \vdash \top}$$

$$\frac{\Gamma \vdash P_1}{\Gamma \vdash P_1 \vee P_2} \qquad \frac{\Gamma \vdash P_2}{\Gamma \vdash P_1 \vee P_2}$$

$$\frac{\Gamma \vdash P_1 \vee P_2 \qquad P_1, \Gamma \vdash P \qquad P_2, \Gamma \vdash P}{\Gamma \vdash P}$$

$$\frac{\Gamma \vdash P_1 \qquad \Gamma \vdash P_2}{\Gamma \vdash P_1 \wedge P_2} \qquad \frac{\Gamma \vdash P_1 \wedge P_2}{\Gamma \vdash P_1} \qquad \frac{\Gamma \vdash P_1 \wedge P_2}{\Gamma \vdash P_2}$$

$$\frac{\Gamma \vdash P_1 \to P_2 \qquad \Gamma \vdash P_1}{\Gamma \vdash P_2} \qquad \frac{P_1, \Gamma \vdash P_2}{\Gamma \vdash P_1 \to P_2}$$

This means, if we can construct a program of a certain type, we have also created a constructive proof of a certain proposition.

The Curry-Howard Correspondence

This correspondence goes by many names, but is usually attributed to Haskell Curry and William Howard.
It is a *very deep* result:

Programming	Logic
Types	Propositions
Programs	Proofs
Evaluation	Proof Simplification

The Curry-Howard Correspondence

This correspondence goes by many names, but is usually attributed to Haskell Curry and William Howard.
It is a *very deep* result:

Programming	Logic
Types	Propositions
Programs	Proofs
Evaluation	Proof Simplification

It turns out, no matter what logic you want to define, there is always a corresponding λ-calculus, and vice versa.

Constructive Logic	Typed λ-Calculus
Classical Logic	Continuations
Modal Logic	Monads
Linear Logic	Linear Types, Session Types
Separation Logic	Region Types

Examples

Example (Commutativity of Conjunction)

$$\begin{aligned} andComm &: A \times B \to B \times A \\ andComm &= \lambda p.\, (\mathsf{snd}\ p, \mathsf{fst}\ p) \end{aligned}$$

This proves $A \wedge B \to B \wedge A$.

Examples

Example (Commutativity of Conjunction)

$$\begin{aligned} andComm &: A \times B \to B \times A \\ andComm &= \lambda p.\ (\mathrm{snd}\ p, \mathrm{fst}\ p) \end{aligned}$$

This proves $A \wedge B \to B \wedge A$.

Example (Transitivity of Implication)

$$\begin{aligned} transitive &: (A \to B) \to (B \to C) \to (A \to C) \\ transitive &= \lambda f\ \lambda g\ \lambda x.\ g\ (f\ x) \end{aligned}$$

Transitivity of implication is just function composition.

Caveats

All functions we define have to be total and terminating.
Otherwise we get an *inconsistent* logic that lets us prove false things:

$$\begin{aligned} &\mathit{proof}_1 : \mathrm{P} = \mathrm{NP} \\ &\mathit{proof}_1 = \mathit{proof}_1 \end{aligned}$$

$$\begin{aligned} &\mathit{proof}_2 : \mathrm{P} \neq \mathrm{NP} \\ &\mathit{proof}_2 = \mathit{proof}_2 \end{aligned}$$

This is why Agda and Idris avoid adding **fix**.
Most common calculi correspond to constructive logic, not classical ones, so principles like the law of excluded middle or double negation elimination do not hold:

$$\neg\neg P \to P$$