7. Fixed Points

UNIVERSITÄT
BERN

Roadmap

- Representing Numbers
- Recursion and the Fixed-Point Combinator
- The typed lambda calculus
- The polymorphic lambda calculus
- Other calculi

References

> Paul Hudak, *"Conception, Evolution, and Application of Functional Programming Languages,"* ACM Computing Surveys 21/3, Sept. 1989, pp 359-411.

Roadmap

- **Representing Numbers**
- Recursion and the Fixed-Point Combinator
- The typed lambda calculus
- The polymorphic lambda calculus
- Other calculi

Recall these encodings …

$$
\begin{aligned}
\text{True} &\equiv \lambda\, x\, y\, .\, x \\
\text{False} &\equiv \lambda\, x\, y\, .\, y \\
\text{pair} &\equiv (\lambda\, x\, y\, z\, .\, z\, x\, y) \\
(x, y) &\equiv \text{pair}\ x\ y \\
\text{first} &\equiv (\lambda\, p\, .\, p\ \text{True}\,) \\
\text{second} &\equiv (\lambda\, p\, .\, p\ \text{False}\,)
\end{aligned}
$$

Representing Numbers

There is a “standard encoding” of natural numbers into the lambda calculus:

Define:

$$0 \equiv (\lambda\, x \,.\, x\,)$$

$$\text{succ} \equiv (\lambda\, n \,.\, (\text{False}, n)\,)$$

then:

$$1 \equiv \text{succ } 0 \quad \to \quad (\text{False}, 0)$$

$$2 \equiv \text{succ } 1 \quad \to \quad (\text{False}, 1)$$

$$3 \equiv \text{succ } 2 \quad \to \quad (\text{False}, 2)$$

$$4 \equiv \text{succ } 3 \quad \to \quad (\text{False}, 3)$$

Working with numbers

We can define simple functions to work with our numbers.

Consider:

iszero	≡	first
pred	≡	second

then:

iszero 1	=	first (False, 0)	→	False
iszero 0	=	(λ p . p True) (λ x . x)	→	True
pred 1	=	second (False, 0)	→	0

- *What happens when we apply pred 0? What does this mean?*

Roadmap

- Representing Numbers
- **Recursion and the Fixed-Point Combinator**
- The typed lambda calculus
- The polymorphic lambda calculus
- Other calculi

Recursion

Suppose we want to define *arithmetic operations* on our lambda-encoded numbers.

In Haskell we can program:

```
plus n m
    | n == 0     = m
    | otherwise = plus (n-1) (m+1)
```

so we might try to "define":

$$plus \equiv \lambda\, n\, m\, .\; iszero\ n\ m\ (\, plus\ (\, pred\ n\,)\ (\, succ\ m\,)\,)$$

Unfortunately this is *not a definition*, since we are trying to *use plus before it is defined*. I.e, plus is *free* in the "definition"!

Recursive functions as fixed points

We can obtain a closed expression by *abstracting over plus:*

rplus ≡ λ plus n m . iszero n
m
(plus (pred n) (succ m))

rplus takes as its *argument* the actual plus function to use and returns as its result a definition of that function in terms of itself. In other words, if **fplus** is the function we want, then:

rplus fplus ↔ fplus

I.e., we are searching for a *fixed point* of rplus ...

Fixed Points

A fixed point of a function f is a value p such that f p = p.

Examples:

```
fact 1     = 1
fact 2     = 2
fib 0      = 0
fib 1      = 1
```

Fixed points are not always “well-behaved”:

```
succ n = n + 1
```

- *What is a fixed point of succ?*

Fixed Point Theorem

Theorem:

Every lambda expression e has a fixed point p such that (e p) ↔ p.

Proof:

Let: Y ≡ λ f . (λ x . f (x x)) (λ x . f (x x))

Now consider:

p ≡ Y e → (λ **x.** e (x x)) **(λ x . e (x x))**

→ e ((λ x . e (x x)) (λ x . e (x x)))

= e p

So, the "magical Y combinator" can always be used to find a fixed point of an *arbitrary* lambda expression.

∀e: Y e ↔ e (Y e)

How does Y work?

Recall the non-terminating expression

$$\Omega = (\lambda\ x\ .\ x\ x)\ (\lambda\ x\ .\ x\ x)$$

Ω loops endlessly without doing any productive work.

Note that (x x) represents the body of the “loop”.

We simply define Y to take an *extra parameter f*, and *put it into the loop*, passing it the body as an argument:

$$Y \equiv \lambda\ f\ .\ (\lambda\ x\ .\ f\ (x\ x))\ (\lambda\ x\ .\ f\ (x\ x))$$

So Y just inserts some productive work into the body of Ω

Using the Y Combinator

Consider:

$$f \equiv \lambda x.\ True$$

then:

$$\begin{aligned} Y\ f &\rightarrow f\ (Y\ f) && \textit{by FP theorem} \\ &= (\lambda x.\ True)\ (Y\ f) \\ &\rightarrow True \end{aligned}$$

Consider:

$$\begin{aligned} Y\ succ &\rightarrow succ\ (Y\ succ) && \textit{by FP theorem} \\ &\rightarrow (False,\ (Y\ succ)) \end{aligned}$$

- *What are succ and pred of (False, (Y succ))? What does this represent?*

Recursive Functions are Fixed Points

We seek a fixed point of:

$$\text{rplus} \equiv \lambda \text{ plus n m . iszero n m (plus (pred n) (succ m))}$$

By the Fixed Point Theorem, we simply take:

$$\text{plus} \leftrightarrow \text{Y rplus}$$

Since this guarantees that:

$$\text{rplus plus} \leftrightarrow \text{plus}$$

as desired!

Unfolding Recursive Lambda Expressions

plus 1 1 = (**Y rplus**) 1 1

→ **rplus plus 1 1** *(NB: fp theorem)*

→ **iszero 1** 1 (plus (pred 1) (succ 1))

→ **False 1 (plus (pred 1) (succ 1))**

→ **plus** (pred 1) (succ 1)

→ **rplus plus (pred 1) (succ 1)**

→ iszero (**pred 1**) (succ 1)

(plus (pred (pred 1)) (succ (succ 1)))

→ **iszero 0** (succ 1) (...)

→ **True (succ 1) (...)**

→ **succ 1**

→ 2

Roadmap

> Representing Numbers
> Recursion and the Fixed-Point Combinator
> **The typed lambda calculus**
> The polymorphic lambda calculus
> Other calculi

The Typed Lambda Calculus

There are many variants of the lambda calculus.

The typed lambda calculus just *decorates terms with type annotations:*

Syntax:

$$e ::= x^{\tau} \mid e_1^{\tau 2\rightarrow \tau 1}\ e_2^{\tau 2} \mid (\lambda\, x^{\tau 2}.e^{\tau 1})^{\tau 2\rightarrow \tau 1}$$

Operational Semantics:

$$\lambda\, x^{\tau 2} .\, e^{\tau 1} \quad \Leftrightarrow \quad \lambda\, y^{\tau 2} .\, [y^{\tau 2}/x^{\tau 2}]\, e^{\tau 1} \qquad y^{\tau 2} \textit{ not free in } e^{\tau 1}$$

$$(\lambda\, x^{\tau 2} .\, e_1^{\tau 1})\, e_2^{\tau 2} \quad \Rightarrow \quad [e_2^{\tau 2}/x^{\tau 2}]\, e_1^{\tau 1}$$

$$\lambda\, x^{\tau 2} .\, (e^{\tau 1}\, x^{\tau 2}) \quad \Rightarrow \quad e^{\tau 1} \qquad x^{\tau 2} \textit{ not free in } e^{\tau 1}$$

Example:

$$\text{True} \equiv (\lambda\, x^{A} .\, (\lambda\, y^{B} .\, x^{A})^{B\rightarrow A})^{A \rightarrow (B\rightarrow A)}$$

Roadmap

> Representing Numbers
> Recursion and the Fixed-Point Combinator
> The typed lambda calculus
> **The polymorphic lambda calculus**
> Other calculi

The Polymorphic Lambda Calculus

Polymorphic functions like “map” cannot be typed in the typed lambda calculus!

Need *type variables* to capture polymorphism:

β reduction (ii):

$$(\lambda\ x^{\nu}\ .\ e_1^{\tau 1})\ e_2^{\tau 2} \Rightarrow [\tau 2/\nu]\ [e_2^{\tau 2}/x^{\nu}]\ e_1^{\tau 1}$$

Example:

$$\begin{aligned} \text{True} &\equiv (\lambda\ x^{\alpha}.\ (\lambda\ y^{\beta}\ .\ x^{\alpha})^{\beta\to\alpha})^{\alpha\to(\beta\to\alpha)} \\ \text{True}^{\alpha\to(\beta\to\alpha)}\ a^{A}\ b^{B} &\to (\lambda\ y^{\beta}.\ a^{A}\ )^{\beta\to A}\ b^{B} \\ &\to a^{A} \end{aligned}$$

Hindley-Milner Polymorphism

Hindley-Milner polymorphism (i.e., that adopted by ML and Haskell) works by inferring the type annotations for a slightly restricted subcalculus: polymorphic functions.

If:

```
doubleLen len len' xs ys = (len xs) + (len' ys)
```

then

```
doubleLen length length "aaa" [1,2,3]
```

is ok, but if

```
doubleLen' len xs ys = (len xs) + (len ys)
```

then

```
doubleLen' length "aaa" [1,2,3]
```

is a type error since the argument `len` *cannot be assigned a unique type!*

Polymorphism and self application

Even the polymorphic lambda calculus is not powerful enough to express certain lambda terms.

Recall that both Ω and the Y combinator make use of "self application":

$$\Omega = (\lambda\ x\ .\ \mathbf{x\ x}\ )\ (\lambda\ x\ .\ x\ x\ )$$

- *What type annotation would you assign to (λ x . x x)?*

Built-in recursion with letrec AKA def AKA μ

- Most programming languages provide direct support for recursively-defined functions (avoiding the need for Y)

$$\boxed{(\textbf{def}\ f.E)\ e \rightarrow E\ [(\textbf{def}\ f.E)\ /\ f]\ e}$$

(def plus. λ n m . iszero n m (plus (pred n) (succ m))) 2 3
→ (λ n m . iszero n m ((def plus. …) (pred n) (succ m))) 2 3
→ (iszero 2 3 ((def plus. …) (pred 2) (succ 3)))
→ ((def plus. …) (pred 2) (succ 3))
→ …

Roadmap

- Representing Numbers
- Recursion and the Fixed-Point Combinator
- The typed lambda calculus
- The polymorphic lambda calculus
- **Other calculi**

Featherweight Java

Syntax:

$$\begin{array}{lcl} \texttt{CL} & ::= & \texttt{class C extends C } \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \\ \texttt{K} & ::= & \texttt{C}(\overline{\texttt{C}}\ \overline{\texttt{f}})\ \{\texttt{super}(\overline{\texttt{f}});\ \texttt{this}.\overline{\texttt{f}} = \overline{\texttt{f}};\} \\ \texttt{M} & ::= & \texttt{C m}(\overline{\texttt{C}}\ \overline{\texttt{x}})\ \{\texttt{return e;}\} \\ \texttt{e} & ::= & \texttt{x} \\ & | & \texttt{e.f} \\ & | & \texttt{e.m}(\overline{\texttt{e}}) \\ & | & \texttt{new C}(\overline{\texttt{e}}) \\ & | & (\texttt{C})\texttt{e} \end{array}$$

Subtyping:

$$\texttt{C} <: \texttt{C}$$

$$\frac{\texttt{C} <: \texttt{D} \qquad \texttt{D} <: \texttt{E}}{\texttt{C} <: \texttt{E}}$$

$$\frac{CT(\texttt{C}) = \texttt{class C extends D } \{\ldots\}}{\texttt{C} <: \texttt{D}}$$

Computation:

$$\frac{fields(\texttt{C}) = \overline{\texttt{C}}\ \overline{\texttt{f}}}{(\texttt{new C}(\overline{\texttt{e}})).\texttt{f}_i \longrightarrow \texttt{e}_i} \quad \text{(R-FIELD)}$$

$$\frac{mbody(\texttt{m}, \texttt{C}) = (\overline{\texttt{x}}, \texttt{e}_0)}{(\texttt{new C}(\overline{\texttt{e}})).\texttt{m}(\overline{\texttt{d}}) \longrightarrow [\overline{\texttt{d}}/\overline{\texttt{x}}, \texttt{new C}(\overline{\texttt{e}})/\texttt{this}]\texttt{e}_0} \quad \text{(R-INVK)}$$

$$\frac{\texttt{C} <: \texttt{D}}{(\texttt{D})(\texttt{new C}(\overline{\texttt{e}})) \longrightarrow \texttt{new C}(\overline{\texttt{e}})} \quad \text{(R-CAST)}$$

Expression typing:

$$\Gamma \vdash \texttt{x} \in \Gamma(\texttt{x}) \quad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash \texttt{e}_0 \in \texttt{C}_0 \qquad fields(\texttt{C}_0) = \overline{\texttt{C}}\ \overline{\texttt{f}}}{\Gamma \vdash \texttt{e}_0.\texttt{f}_i \in \texttt{C}_i} \quad \text{(T-FIELD)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{e}_0 \in \texttt{C}_0 \\ mtype(\texttt{m}, \texttt{C}_0) = \overline{\texttt{D}} \to \texttt{C} \\ \Gamma \vdash \overline{\texttt{e}} \in \overline{\texttt{C}} \qquad \overline{\texttt{C}} <: \overline{\texttt{D}}\end{array}}{\Gamma \vdash \texttt{e}_0.\texttt{m}(\overline{\texttt{e}}) \in \texttt{C}} \quad \text{(T-INVK)}$$

$$\frac{\begin{array}{c}fields(\texttt{C}) = \overline{\texttt{D}}\ \overline{\texttt{f}} \\ \Gamma \vdash \overline{\texttt{e}} \in \overline{\texttt{C}} \qquad \overline{\texttt{C}} <: \overline{\texttt{D}}\end{array}}{\Gamma \vdash \texttt{new C}(\overline{\texttt{e}}) \in \texttt{C}} \quad \text{(T-NEW)}$$

$$\frac{\Gamma \vdash \texttt{e}_0 \in \texttt{D} \qquad \texttt{D} <: \texttt{C}}{\Gamma \vdash (\texttt{C})\texttt{e}_0 \in \texttt{C}} \quad \text{(T-UCAST)}$$

$$\frac{\Gamma \vdash \texttt{e}_0 \in \texttt{D} \qquad \texttt{C} <: \texttt{D} \qquad \texttt{C} \neq \texttt{D}}{\Gamma \vdash (\texttt{C})\texttt{e}_0 \in \texttt{C}} \quad \text{(T-DCAST)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{e}_0 \in \texttt{D} \qquad \texttt{C} \not<: \texttt{D} \qquad \texttt{D} \not<: \texttt{C} \\ \textit{stupid warning}\end{array}}{\Gamma \vdash (\texttt{C})\texttt{e}_0 \in \texttt{C}} \quad \text{(T-SCAST)}$$

Method typing:

$$\frac{\begin{array}{c}\overline{\texttt{x}} : \overline{\texttt{C}}, \texttt{this} : \texttt{C} \vdash \texttt{e}_0 \in \texttt{E}_0 \qquad \texttt{E}_0 <: \texttt{C}_0 \\ CT(\texttt{C}) = \texttt{class C extends D } \{\ldots\} \\ override(\texttt{m}, \texttt{D}, \overline{\texttt{C}} \to \texttt{C}_0)\end{array}}{\texttt{C}_0\ \texttt{m}\ (\overline{\texttt{C}}\ \overline{\texttt{x}})\ \{\texttt{return e}_0;\}\ \texttt{OK IN C}}$$

Class typing:

$$\frac{\begin{array}{c}\texttt{K} = \texttt{C}(\overline{\texttt{D}}\ \overline{\texttt{g}}, \overline{\texttt{C}}\ \overline{\texttt{f}})\ \{\texttt{super}(\overline{\texttt{g}});\ \texttt{this}.\overline{\texttt{f}} = \overline{\texttt{f}};\} \\ fields(\texttt{D}) = \overline{\texttt{D}}\ \overline{\texttt{g}} \qquad \overline{\texttt{M}}\ \texttt{OK IN C}\end{array}}{\texttt{class C extends D } \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\}\ \texttt{OK}}$$

Used to prove that generics could be added to Java without breaking the type system.

Igarashi, Pierce and Wadler, *"Featherweight Java: a minimal core calculus for Java and GJ", OOPSLA '99*
doi.acm.org/10.1145/320384.320395

Other Calculi

Many calculi have been developed to study the semantics of programming languages.

Object calculi: model *inheritance and subtyping ..*
- lambda calculi with records

Process calculi: model *concurrency and communication*
- CSP, CCS, pi calculus, CHAM, blue calculus

Distributed calculi: model *location and failure*
- ambients, join calculus

A quick look at the π calculus

new channel

outpu
t

concurrency

inpu
t

$$\nu(x)(\underline{\mathbf{x}}\mathbf{\langle z\rangle}.0 \mid \mathbf{x(y)}.\underline{y}\langle x\rangle.x(y).0) \mid z(v).\underline{v}\langle v\rangle.0$$

$$\rightarrow \nu(x)(0 \mid \underline{\mathbf{z}}\mathbf{\langle x\rangle}.x(y).0) \mid \mathbf{z(v)}.\underline{v}\langle v\rangle.0$$

$$\rightarrow \nu(x)(0 \mid \mathbf{x(y)}.0 \mid \underline{\mathbf{x}}\mathbf{\langle x\rangle}.0)$$

$$\rightarrow \nu(x)(0 \mid 0 \mid 0)$$

en.wikipedia.org/wiki/Pi_calculus

What you should know!

- *Why isn't it possible to express recursion directly in the lambda calculus?*
- *What is a fixed point? Why is it important?*
- *How does the typed lambda calculus keep track of the types of terms?*
- *How does a polymorphic function differ from an ordinary one?*

Can you answer these questions?

- *How would you model negative integers in the lambda calculus? Fractions?*
- *Is it possible to model real numbers? Why, or why not?*
- *Are there more fixed-point operators other than Y?*
- *How can you be sure that unfolding a recursive expression will terminate?*
- *Would a process calculus be Church-Rosser?*

License

http://creativecommons.org/licenses/by-sa/3.0/

creative commons

COMMONS DEED

Attribution-ShareAlike 3.0 Unported

You are free:

to Share — to copy, distribute and transmit the work

to Remix — to adapt the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.

Any of the above conditions can be waived if you get permission from the copyright holder.

Nothing in this license impairs or restricts the author's moral rights.