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 ...

True	≡	λ x y . x
False	≡	λ x y . y
pair	≡	(λ x y z . z x y)
(x, y)	≡	pair x y
first	≡	(λ p . p True)
second	≡	(λ p . p False)

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:

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

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

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

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

Working with numbers

We can define simple functions to work with our numbers.

Consider:

$$\text{iszero} \equiv \text{first}$$

$$\text{pred} \equiv \text{second}$$

then:

$$\text{iszero } 1 = \text{first (False, 0)} \rightarrow \text{False}$$

$$\text{iszero } 0 = (\lambda\, p \,.\, p \text{ True })\ (\lambda\, x \,.\, x\ ) \rightarrow \text{True}$$

$$\text{pred } 1 = \text{second (False, 0)} \rightarrow 0$$

- *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":

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

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) $\leftrightarrow$ p.

Proof:

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

Now consider:

$$
\begin{aligned}
p \equiv Y e &\rightarrow (\lambda \mathbf{x}. e (x x)) \boldsymbol{(\lambda x . e (x x))} \\
&\rightarrow e ((\lambda x . e (x x)) (\lambda x . e (x x))) \\
&= e p
\end{aligned}
$$

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

$$\forall e: Y e \leftrightarrow e (Y e)$$

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:

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

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:

$\lambda\, x^{\tau 2} . e^{\tau 1}$	$\Leftrightarrow$	$\lambda\, y^{\tau 2} . [y^{\tau 2}/x^{\tau 2}]\, e^{\tau 1}$	*$y^{\tau 2}$ not free in $e^{\tau 1}$*
$(\lambda\, x^{\tau 2} . e_1^{\tau 1})\, e_2^{\tau 2}$	$\Rightarrow$	$[e_2^{\tau 2}/x^{\tau 2}]\, e_1^{\tau 1}$	
$\lambda\, x^{\tau 2} . (e^{\tau 1}\, x^{\tau 2})$	$\Rightarrow$	$e^{\tau 1}$	*$x^{\tau 2}$ 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:

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

Subtyping:

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

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

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

Computation:

$$\frac{\mathit{fields}(\texttt{C}) = \bar{\texttt{C}}\ \bar{\texttt{f}}}{\texttt{(new C(}\bar{\texttt{e}}\texttt{)).f}_i \longrightarrow \texttt{e}_i} \qquad \text{(R-Field)}$$

$$\frac{\mathit{mbody}(\texttt{m},\texttt{C}) = (\bar{\texttt{x}}, \texttt{e}_0)}{\texttt{(new C(}\bar{\texttt{e}}\texttt{)).m(}\bar{\texttt{d}}\texttt{)} \longrightarrow [\bar{\texttt{d}}/\bar{\texttt{x}}, \texttt{new C(}\bar{\texttt{e}}\texttt{)/this}]\texttt{e}_0} \qquad \text{(R-Invk)}$$

$$\frac{\texttt{C} <: \texttt{D}}{\texttt{(D)(new C(}\bar{\texttt{e}}\texttt{))} \longrightarrow \texttt{new C(}\bar{\texttt{e}}\texttt{)}} \qquad \text{(R-Cast)}$$

Expression typing:

$$\Gamma \vdash \texttt{x} \in \Gamma(\texttt{x}) \qquad \text{(T-Var)}$$

$$\frac{\Gamma \vdash \texttt{e}_0 \in \texttt{C}_0 \qquad \mathit{fields}(\texttt{C}_0) = \bar{\texttt{C}}\ \bar{\texttt{f}}}{\Gamma \vdash \texttt{e}_0\texttt{.f}_i \in \texttt{C}_i} \qquad \text{(T-Field)}$$

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

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

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

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

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

Method typing:

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

Class typing:

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

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/