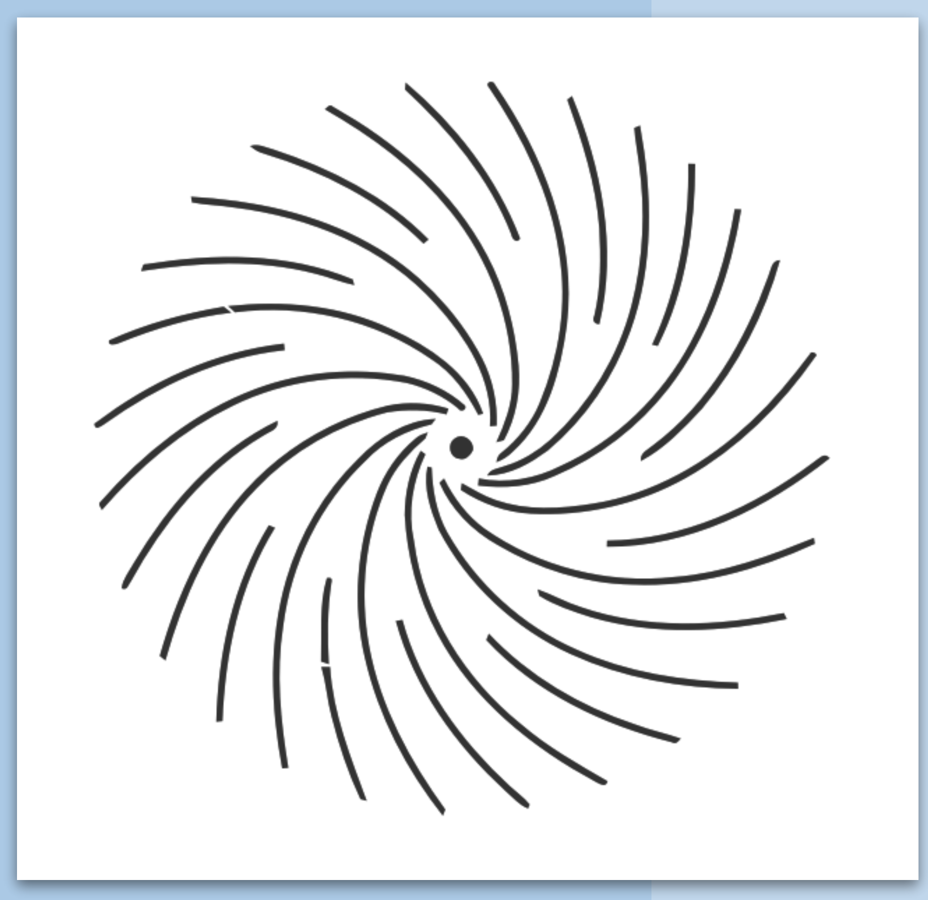
UNIVERSITÄT
BERN

6. Fixed Points

Oscar Nierstrasz

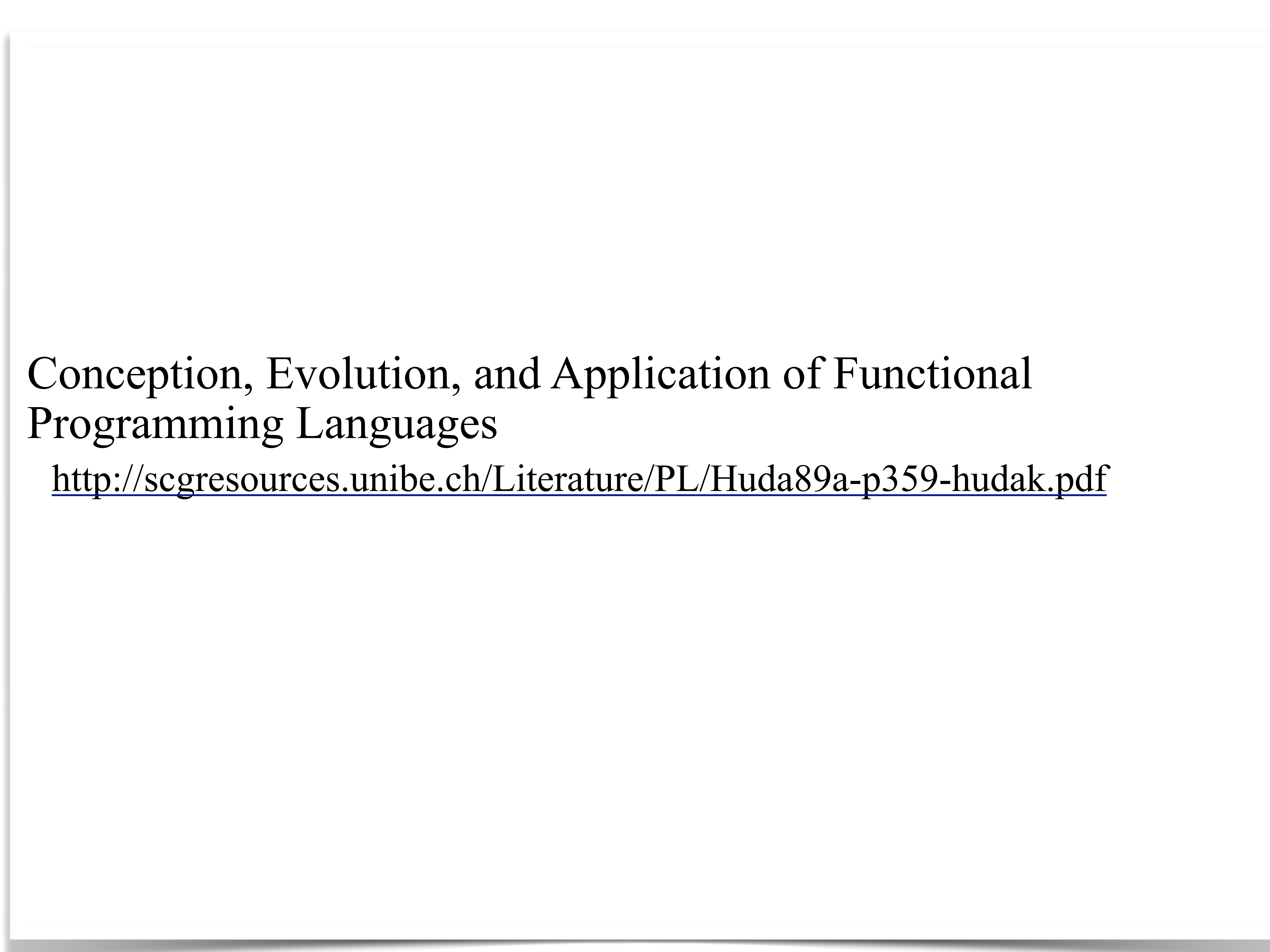


Roadmap

- 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.


Conception, Evolution, and Application of Functional Programming Languages
http://scgresources.unibe.ch/Literature/PL/Huda89a-p359-hudak.pdf

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:

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

$$succ \equiv (\lambda\, n\, .\, (False,\, n)\,)$$

then:

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

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

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

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

Note how all positive integers are encoded as a pair *(b,n)*, where *b* is a Boolean indicating whether the number is zero or not (i.e., always *False* for positive integers), and *n* represents the number's predecessor. In other words, the number *n+1* is represented by *(False,n)*, which is computed by the function application *succ n*.

The exception is the integer value 0, which is represented by the identify function.

Now the question is, how do we make sense of this representation of 0?

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?*

To work with our numbers, we need to do at least two further things: check if a number is zero or not, and compute its predecessor.

If we define *iszero* as the function *first* (seen earlier) then we see that it works correctly both for positive integers and the value 0, returning *False* for the former and *True* for the latter.

(Ok, it's a trick, but it's a cool trick!)

Similarly, if we define *pred* as *second*, it correctly returns the predecessor of all positive integers.

What, however, is the meaning of *pred 0* ?

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"!

The problem is that ≡ is not part of the syntax of the lambda calculus: we have no way to define a "global function", but can only work with anonymous lambdas. The only way we can bind a name to a function is using a lambda itself. For example, if we want to evaluate *not True*, then we need to first bind the names *True* and *False* using lambdas, then *not* using *True* and *False*, and finally evaluate *not True*:

(λ True False .
  (λ not . **not True**)
  (λ b.b False True)
) (λ x y.x) (λ x y.y)

With *plus* this trick will not work, since we would need to bind plus before we could use it, which is impossible!

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:

$$\text{rplus fplus} \leftrightarrow \text{fplus}$$

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

The trick here is to *abstract away from plus*, i.e., to consider it as a *parameter* to the function *plus* we are defining. Although this seems circular, it is not. We are saying how to *define plus*, if we are given an *implementation* of *plus*. These are two different things.

Now the question is how we can turn this definition into the implementation that we need!

The missing implementation is a “fixed point” of the abstraction rplus: if we supply it as an argument, we get the same value back as a result.

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:

$$p \equiv Y\ e \rightarrow (\lambda x. e (x\ x)) \mathbf{(\lambda x . e (x\ x))}$$

$$\rightarrow e ((\lambda x . e (x\ x)) (\lambda x . e (x\ x)))$$

$$= e\ p$$

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

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

The “magical” Y combinator will take any lambda expression *e*, and yield a value *Y e*, which, when passed to *e* as an argument, returns a result that is *equivalent* (i.e., convertible with reductions and conversions) to *Y e*. Y e is therefore a *fixpoint* (AKA “fixed point”) of *e*.

Y is “magical” in the sense that it is not immediately obvious why it should work, but it is not too hard to see why ...

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

Note how Ω just loops around endlessly without doing anything useful. The expression *(x x)* represents this loop.

Recall that *Y e* is the fixpoint we seek. We define Y just like Ω, but we add an extra parameter *f* (which will be bound to *e*), and we pass it the result of the loop, i.e., *(x x)*. Now it is up to *e* to decide when to terminate the loop!

If *e* is defined recursively, it will loop as often as it needs to, and then end with the result it is supposed to compute. By the trick of self-application in *(x x)*, it can get as many copies of the fixpoint as it needs, but it no longer has to loop endlessly.

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?*

Recall that the fixpoint theorem states that

$$Y\ e \rightarrow e\ (Y\ e)$$

We simply apply this rewriting and evaluate the resulting expression. Note that for any constant function (λ x. k) (where k is a constant):

$$Y\ (\lambda\ x.\ k) \rightarrow (\lambda\ x.\ k)\ (Y\ (\lambda\ x.\ k)) \rightarrow k$$

In other words, the fixpoint of a constant function is that constant, which is as we expect.

Recursive Functions are Fixed Points

We seek a fixed point of:

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

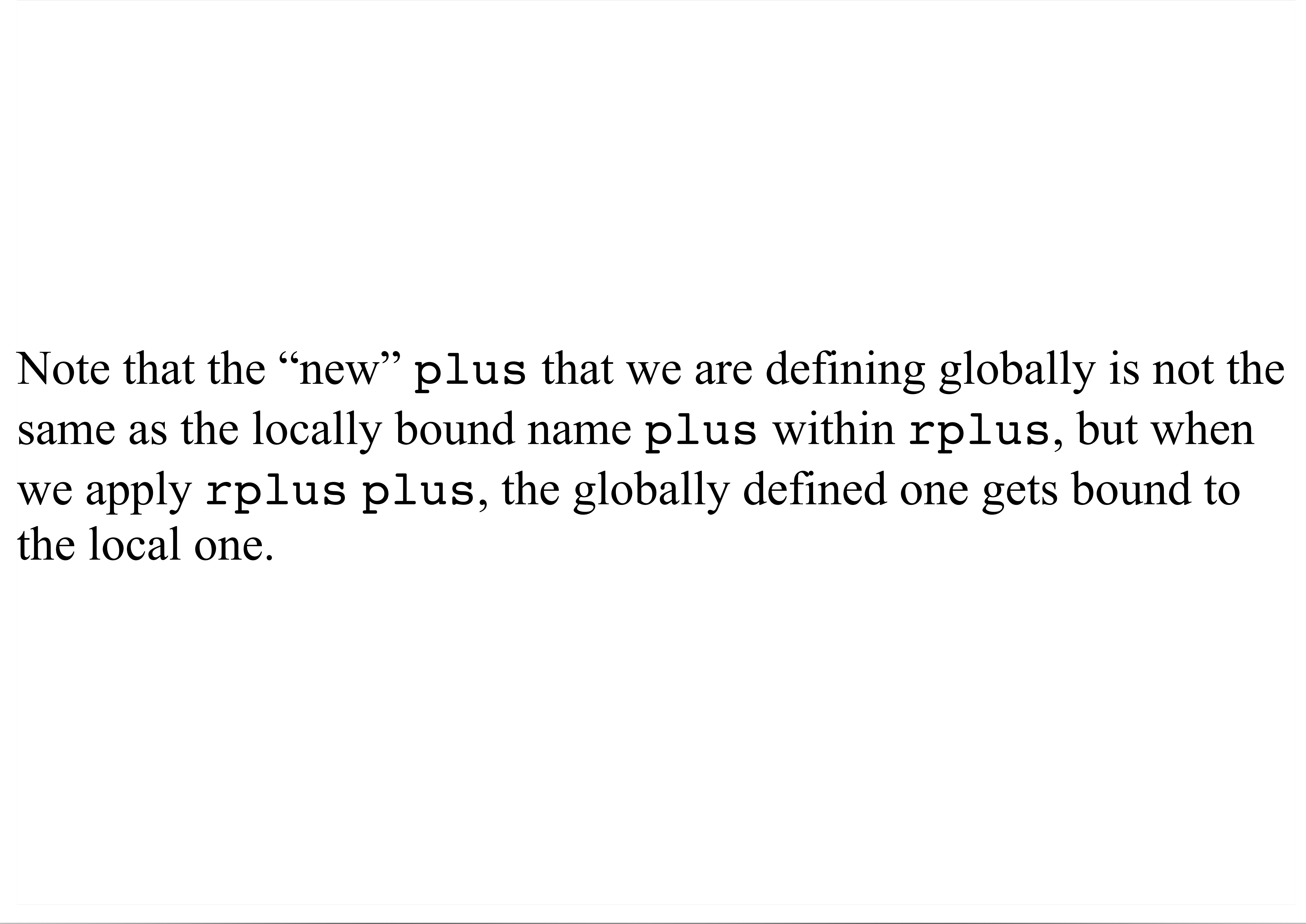
By the Fixed Point Theorem, we simply define:

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

Since this guarantees that:

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

as desired!



Note that the “new” `plus` that we are defining globally is not the same as the locally bound name `plus` within `rplus`, but when we apply `rplus plus`, the globally defined one gets bound to the local one.

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

Here we can see how, like Ω, *Y rplus* loops repeatedly. The difference is that we can do productive work at each step, and decide when to terminate the loop with the *iszero* test.

Note that we use the fixpoint theorem also in lines 5 and 6 to rewrite plus as rplus plus.

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 <u>typed lambda calculus</u> 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:

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

This looks complicated due to the messy annotations, but the operational semantics is exactly the same as in the untyped lambda calculus. The α, β and η rules work just as before, they just keep track of additional type information, encoding for each expression and subexpression what its type is.

Note that this is a *monomorphic type system*: every value as a unique, concrete type, not a variable type. The function

$\lambda\, x^A . x^A$

Takes arguments only of type A, not any other type!

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:

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

NB: some superscripts have been left off for readability!

Type variables solve the problem with the monomorphic typed lambda calculus. Now we can write:

$\lambda\, x^{\alpha} .\, x^{\alpha}$

where α is a *type variable*. This function will accept an argument of any type A, which will then be *bound* to α. In other words, if this function accepts an argument of type A, it will return an value x of the *same* type A.

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)?*

Note that there is *no reasonable type* that we can assign to an expression like $x\ x$. It is therefore impossible to express Y in the typed lambda calculus, so there is no way to express recursion, and consequently, one can only express *terminating* programs using it!

For the same reason, we cannot write Y in Haskell (try it!).

Built-in recursion with letrec AKA def AKA μ

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

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

(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))
→ …

The "def" feature allows you to assign a name (such as *f*) to a lambda expression *E*, where *f* may be used recursively within *E*. The β reduction rule for this extended lambda calculus simply propagates the definition to the recursive uses of *f* within *E* (exactly as Y does!).

Roadmap

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

Featherweight Java

Syntax:

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

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:

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

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

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

Expression typing:

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

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

$$\frac{\Gamma \vdash \texttt{e}_0 \in \texttt{C}_0 \quad mtype(\texttt{m}, \texttt{C}_0) = \bar{\texttt{D}} \rightarrow \texttt{C} \quad \Gamma \vdash \bar{\texttt{e}} \in \bar{\texttt{C}} \quad \bar{\texttt{C}} <: \bar{\texttt{D}}}{\Gamma \vdash \texttt{e}_0.\texttt{m}(\bar{\texttt{e}}) \in \texttt{C}} \quad \text{(T-INVK)}$$

$$\frac{fields(\texttt{C}) = \bar{\texttt{D}}\ \bar{\texttt{f}} \quad \Gamma \vdash \bar{\texttt{e}} \in \bar{\texttt{C}} \quad \bar{\texttt{C}} <: \bar{\texttt{D}}}{\Gamma \vdash \texttt{new C}(\bar{\texttt{e}}) \in \texttt{C}} \quad \text{(T-NEW)}$$

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

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

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

Method typing:

$$\frac{\bar{\texttt{x}} : \bar{\texttt{C}}, \texttt{this} : \texttt{C} \vdash \texttt{e}_0 \in \texttt{E}_0 \quad \texttt{E}_0 <: \texttt{C}_0 \quad CT(\texttt{C}) = \texttt{class C extends D } \{\ldots\} \quad override(\texttt{m}, \texttt{D}, \bar{\texttt{C}} \rightarrow \texttt{C}_0)}{\texttt{C}_0\ \texttt{m}\ (\bar{\texttt{C}}\ \bar{\texttt{x}})\ \{\texttt{return e}_0;\}\ \texttt{OK IN C}}$$

Class typing:

$$\frac{\texttt{K} = \texttt{C}(\bar{\texttt{D}}\ \bar{\texttt{g}}, \bar{\texttt{C}}\ \bar{\texttt{f}})\ \{\texttt{super}(\bar{\texttt{g}});\ \texttt{this}.\bar{\texttt{f}} = \bar{\texttt{f}};\} \quad fields(\texttt{D}) = \bar{\texttt{D}}\ \bar{\texttt{g}} \quad \bar{\texttt{M}}\ \texttt{OK IN C}}{\texttt{class C extends D } \{\bar{\texttt{C}}\ \bar{\texttt{f}};\ \texttt{K}\ \bar{\texttt{M}}\}\ \texttt{OK}}$$

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

Featherweight Java is an object calculus inspired by the lambda calculus. Most of the rules have a precondition (over the horizontal line) and a conclusion (below the line). There are rules for dynamic semantics (reductions) and static semantics (type rules).

The β reduction rule is called R-INVK. It says, if method `m` in class `C` takes an argument array *x* and has a body e_0, then an invocation of a method m with argument array *d* on an object `(new C(e))` reduces to e_0, with arguments *d* replacing *x* and `(new C(e))` replacing `this`. (Pretty much like in the lambda calculus.

Featherweight Java is especially interesting because (1) it was used to prove some defects in the Java type system, and (2) it showed that the type system could be extended to incorporate generics without breaking existing code!

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

output

concurrency

input

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

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

$$\rightarrow \nu(x)(0 \mid \mathbf{x(y)}.0 \mid \underline{\mathbf{x}}\mathbf{<x>}.0)$$

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

en.wikipedia.org/wiki/Pi_calculus

1. The value `z` is output along channel `x`, to the process `x(y).y<x>...`, causing `y` to be bound to `z`.
2. The value `x` is output along channel `z` to the receiving process at the right, `z(v).v<v>.0`, but since `x` is bound by the scope of `ν(x)(...)`, this forces the receiving process to migrate within the scope of `ν(x)(...)`. Since there is no free `x` in the migrating process, no variable needs o be renamed.
3. Finally, `x` is output along channel `x`, binding it to `y`. The subsequent processes are all dead (`0`).

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?

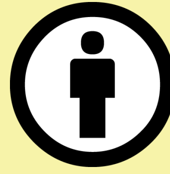
Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:

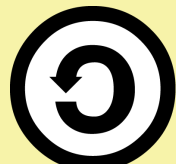
Share — copy and redistribute the material in any medium or format
Adapt — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:



Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.



ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

http://creativecommons.org/licenses/by-sa/4.0/