Fixed Points

Overview:
- Recursion and the fixed-point combinator
- Fixed-point semantics of objects
- The typed lambda-calculus
- The polymorphic lambda calculus

References:
Recursion

Suppose we want to define arithmetic operations on our lambda-encoded numbers (see “Representing Numbers”).

In Haskell we can program:

\[
\text{plus } n \ m = \begin{cases} 
    m & \text{if } n == 0 \\
    \text{plus } (n-1) (m+1) & \text{otherwise}
\end{cases}
\]

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. Although recursion is fundamental to functional programming, it is not primitive in the lambda calculus, so we must find a way to “program” it!
However, we can obtain a closed expression by abstracting over plus:

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

Now, let “fplus” be the actual addition function we want. We must pass it to “rplus” as a parameter before we can perform any additions. But then (rplus fplus) is the function we want. In other words, we are looking for an fplus such that:

\[
\text{rplus fplus} \leftrightarrow \text{fplus}
\]

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

Com S 541
Fixed Points

- In general, a fixed point of a function is a value in the function’s domain, which is mapped to itself by the function. Therefore, a fixed point of a function \( f \) is a value \( p \) such that \( fp = p \).

- Examples:
  - \text{fact 1} = 1
  - \text{fact 2} = 2
  - \text{fib 0} = 0
  - \text{fib 1} = 1
  - \( \lambda x . 27 = 27 \)
  - \( \lambda x . 6 - x = 3 \)

- However, not all functions have exactly one fixed point: “\text{succ } n = n + 1” has none, while “\text{id}” has infinitely many.
Fixed Point Theorem

**Fixed point Theorem:**
For every $F$ there exists a fixed point $X$ such that $FX \leftrightarrow X$.

**Proof:**
Let

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

Now consider:

$$X \equiv Y F \Rightarrow (\lambda x \cdot F (x x)) (\lambda x \cdot F (x x))$$

$$\Rightarrow F ((\lambda x \cdot F (x x)) (\lambda x \cdot F (x x)))$$

$$\Rightarrow F X$$

Therefore, the “$Y$ combinator” can always be used to find a fixed point of an arbitrary lambda expression, if such a fixed point exists.
Using the Y-Combinator

Consider
\[ f \equiv \lambda x. \text{True} \]

Then
\[ Y f \rightarrow f (Y f) \]
\[ = (\lambda x. \text{True}) (Y f) \]
\[ \rightarrow \text{True} \]

Consider
\[ f \equiv \lambda x. x + 1, \text{ where } f \text{ has no fixed point} \]

Then
\[ Y f \rightarrow f (Y f) = (\lambda x. x + 1)(Y f) \rightarrow (Y f) + 1 \rightarrow \ldots \]

Therefore, \( Y \) is also called the “paradoxical combinator”. The problem is that the \( \lambda \)-calculus is a system without semantics [Stoy].

Com S 541
Recursive Functions Are Fixed Points

- We cannot write:
  
  \[ \text{plus} \equiv \lambda \, n \, m \, . \, \text{iszero} \, n \, m \, (\text{plus} \, (\text{pred} \, n) \, (\text{succ} \, m)) \]

  because plus is unbound in the “definition”.

- We can, however, abstract over plus:
  
  \[ \text{rplus} \equiv \lambda \, \text{plus} \, n \, m \, . \, \text{iszero} \, n \, m \, (\text{plus} \, (\text{pred} \, n) \, (\text{succ} \, m)) \]

- Now we seek a lambda expression plus, such that:
  
  \[ \text{rplus} \, \text{plus} \iff \text{plus} \]

- I. e., plus is a fixed point of rplus. By the fixed point theorem, we can take:
  
  \[ \text{plus} \equiv Y \, \text{rplus} \]
Unfolding Recursive Lambda Expressions

Consider:

\[ \text{plus } 1 1 = (Y \ rplus) 1 1 \]
\[ \rightarrow \ rplus \text{ plus } 1 1 \]
\[ \rightarrow \ \text{iszero } 1 1 (\text{plus (pred } 1) (\text{succ } 1)) \]
\[ \rightarrow \ \text{False } 1 (\text{plus (pred } 1) (\text{succ } 1)) \]
\[ \rightarrow \ \text{plus (pred } 1) (\text{succ } 1) \]
\[ \rightarrow \ rplus \text{ plus (pred } 1) (\text{succ } 1) \]
\[ \rightarrow \ \text{iszero (pred } 1) (\text{succ } 1) (\text{plus (pred (pred } 1)) (\text{succ (succ } 1))) \]
\[ \rightarrow \ \text{iszero } 0 (\text{succ } 1) (...) \]
\[ \rightarrow \ \text{True (succ } 1) (...) \]
\[ \rightarrow \ \text{succ } 1 \]
\[ \rightarrow \ 2 \]
Self-reference occurs when a structure is defined in terms of itself.

We can consider self-reference as a form of invocation just like client invocation (of a function or of a method of an object).
Fixed-point Semantics

- Fixed-point semantics of recursive programs provides the mathematical setting for an inheritance model for an object-oriented language.

- Consider the following definition of the factorial function:

\[
\text{fact} = \lambda n \cdot \text{if } n = 1 \text{ then } 1 \text{ else } n \ast \text{fact}(n - 1)
\]

The identifier fact is equated with a function definition, in which “fact” appears. Thus the definition is self-referential. The use of the name fact to represent the self-reference is however just a syntactic convention. In object-oriented languages we commonly use the special symbol self:

\[
\text{fact} = \lambda n \cdot \text{if } n = 1 \text{ then } 1 \text{ else } n \ast \text{self}(n - 1)
\]
Function Transformation

By using fixed-point technique, recursive definitions can be transformed into a non-recursive form:

\[ \text{FACT} = \lambda \text{self} \cdot \lambda n . \begin{cases} 1 & \text{if } n = 1 \\ n \times \text{self}(n-1) & \text{else} \end{cases} \]

\text{FACT} is a functional, or a mapping from functions to functions. The formal parameter "self" represents the function to call in order to compute the factorial function. The original definition of \text{fact} can now be given in terms of \text{FACT}:

\[ \text{fact} = \text{FACT}(\text{fact}) \]

But now \text{fact} is defined as a value that is unchanged when \text{FACT} is applied. Such value is called a "fixed point" of \text{FACT}. Under certain conditions, it is possible to compute a unique fixed point, the least fixed point, of any function by using the fixed-point function "fix" that has the following property: if \( f = \text{fix}(F) \), then \( F(f) = f \).
A Pair Constructor

Consider a pair constructor \( \langle l, r \rangle \) and a pair of selectors \textit{left} and \textit{right}:

\[
p = \langle 3, \text{left}(p) + 1 \rangle
\]

This definition is self-referential but not essentially recursive. Furthermore, \( p = \langle 3, 4 \rangle \).

- \( P = \lambda \text{self} . \langle 3, \text{left}(\text{self}) + 1 \rangle \)
- \( p = P(p) \) and \( p = \text{fix}(P) \)
- Then \( p = \langle 3, 4 \rangle \) is the only pair for which \( \langle 3, 4 \rangle = P(\langle 3, 4 \rangle) \).
Generators

Definition:
A function intended to specify a fixed point whose formal parameter represents self-reference is called generator. Thus a generator has the form

\[ G = \lambda \text{self} . \text{body} \]

where self may occur free in body. Intuitively, self-reference is “unbound” in a generator, while self-reference in its fixed point is connected back to the generator.
A Record Generator

Consider the following record generator representing a pair of values where the second depends upon the first:

\[ G = \lambda \text{self}. \left[ \text{base} \rightarrow 7, \text{square} \rightarrow \text{self.base} \times \text{self.base} \right] \]

The fixed point of this generator is the record

\[ m = \text{fix}(G) = \left[ \text{base} \rightarrow 7, \text{square} \rightarrow 49 \right] \]
The motivation for inheritance is an interaction between self-reference and modification:

However, this derivation has not the effect of destructive modification, because self-reference is used to refer to parts of the original structure that should have been modified. Therefore, self-reference in the original must be changed to refer to the modifications. This is the mechanism of inheritance:
Wrappers

Definition:
A wrapper is a function designed to modify a self-referential structure in a self-referential way; it has two parameters, one representing self-reference and the other representing the superstructure being modified. Thus a wrapper is a function of the form

\[ W = \lambda \text{self} . \lambda \text{super} . \text{Body} \]

where self and super may occur free in body.

The application of a wrapper to a generator involves binding together self-reference in the wrapper and the generator, and then applying the wrapper modification to the value of the generator:

\[ W \oplus G = \lambda \text{self} . W(\text{self})(G(\text{self})), \text{ where } \oplus = \lambda a . \lambda b . \lambda s . a(s)(b(s)) \]
Applying Wrappers

The result of applying a wrapper to a generator is a generator. Self-references in the wrapper W and the generator G are bound together though the variable self, signified by the joining of the arrows out of W and G. The arrow from W to G represents the application of W to G: