4.5 Lambda Calculus

Lambda calculus was introduced in 1930s, as a mathematical theory together with a proof calculus aiming at capturing foundationally the important notions of function and function application. Those years were marked by several paradoxes in mathematics and logics. The original motivation of λ-calculus was to provide a foundation of logics and mathematics. Whether λ-calculus indeed provides a strong foundation of mathematics is still open; what is certain is that λ-calculus was and still is a quite successful *theory of computation*. Today, more than 70 years after its birth, λ-calculus and its afferent subjects still fascinates computer scientists, logicians, mathematicians and, certainly, philosophers.

λ-Calculus is a convenient framework to describe and explain many programming language concepts. It formalizes the informal notion of "expression that can be evaluated" as a *λ-term*, or *λ-expression*. More precisely, λ-calculus consists of:

- *Syntax* - used to express λ-terms, or λ-expressions;
- *Proof system* - used to prove λ-expressions equal;
- *Reduction* - used to reduce λ-expressions to equivalent ones.

We will show how λ-calculus can be formalized as an *equational theory* in equational logic (Section 2.4), provided that an important mechanism of *substitution* is available. That means that its syntax can be defined as an algebraic signature; its proof system becomes a special case of equational deduction; and its reduction becomes a special case of rewriting (when certain equations are regarded as rewrite rules). These results support the overall theme underlying this book, that equational logic and rewriting are a strong foundation for describing and explaining programming language concepts. Even though λ-calculus is a special equational theory, it has the merit that it is powerful enough to express most programming language concepts quite naturally. Equational logic gives us, in some sense, "too much freedom" in how to define concepts; its constraints and intuitions are not restrictive enough to impose an immediate mapping of programming language concepts into it. As extensively illustrated in Chapter 3, due to their generality, equational and rewrite logics need *appropriate methodologies* to define languages, while λ-calculus, due to its particularity, gives an immediate means to syntactically encode programming languages and hereby give them a semantics.

As discussed in Section 4.5.6, many programming language concepts, and even entire programming languages, translate relatively naturally into λ-calculus concepts or into λ-expressions. That includes the various types of values and numbers, data structures, recursion, as well as complex language constructs.

4.5.1 Syntax

Syntactically, λ-calculus consists of two syntactic constructs, λ-abstraction and λ-application, and a given infinite set of variables. Let us assume an infinite set of variables, or names, *Var*. Then the syntax of *λ-expressions* is given by the following grammar (in BNF notation):

Exp	::=	*Var*	— variables are λ-expressions
	|	*λVar.Exp*	— *λ-abstraction,* or *nameless function*
	|	*Exp Exp*	— *λ-application*

We will use lower letters x, y, z, etc., to refer to variables, and capital letters E, E', E_1, E_2, etc., to refer to λ-expressions. The following are therefore examples of λ-expressions:

$\lambda x.x$

$\lambda x.xx$

$\lambda x.(fx)(gx)$

$(\lambda x.fx)x$

λ-Expressions of the form $\lambda x.E$ are called *λ-abstractions*, and those of the form E_1E_2 are called *λ-applications*. The former captures the intuition of nameless functions, while the latter that of function applications. To avoid parentheses, we assume that λ-application is left associative and binds tighter than λ-abstraction; for example, $\lambda x.\lambda y.\lambda z.xyz$ is the same as $\lambda x.\lambda y.\lambda z.((xy)z)$.

4.5.2 Free and Bound Variables

Variable occurrences in λ-expressions can be either *free* or *bound*. Given a λ-abstraction $\lambda x.E$, also called a *binding*, then the variable x is said to be *declared* by the λ-abstraction, or that *$\lambda x.E$ binds x in E*; also, E is called the *scope* of the binding. Formally, we define the set $FV(E)$ of *free variables of E* as follows:

- $FV(x) = \{x\}$,
- $FV(E_1E_2) = FV(E_1) \cup FV(E_2)$, and
- $FV(\lambda x.E) = FV(E) \setminus \{x\}$.

Consider the three underlined occurrences of x in the λ-expression $(\lambda\underline{x}.\lambda y.y\underline{x}y)\underline{x}$. The first is called a *binding occurrence of x*, the second a *bound occurrence of x* (this occurrence of x is bound to the binding occurrence of x), and the third a *free occurrence of x*. Expressions E with $FV(E) = \emptyset$ are called *closed* or *combinators*.

4.5.3 Substitution

Evaluation of λ-expressions is *by substitution*. That means that the λ-expression that is passed to a λ-abstraction is *copied as is* at all the bound occurrences of the binding variable. This will be formally defined later. Let us now formalize and discuss the important notion of *substitution*. Intuitively, $E[E'/x]$ represents the λ-expression obtained from E by replacing each free occurrence of x by E'. For notational simplicity, we assume that, syntactically, substitution binds tighter than any λ-calculus construct; as usual, parentheses can be used to enforce desired grouping. Formally, substitution can be defined as follows:

- $y[E'/x] = \begin{cases} E' & \text{if } \ y = x \\ y & \text{if } \ y \neq x \end{cases}$
- $(E_1E_2)[E'/x] = E_1[E'/x]\ E_2[E'/x]$
- $(\lambda x.E)[E'/x] = \lambda x.E$

The tricky part is to define substitution on λ-abstractions of the form $(\lambda y.E)[E'/x]$, where $y \neq x$. That is because E' may contain free occurrences of y; these occurrences of y would become bound by the binding variable y if one simply defined this substitution as $\lambda y.(E[E'/x])$ (and if E had any free occurrences of x), thus violating the intuitive meaning of binding. This phenomenon is called *variable capturing*. Consider, for example, the substitution $(\lambda y.x)[yy/x]$; if one applies the substitution blindly then one gets $\lambda y.yy$, which is most likely *not* what one meant (since $\lambda y.x$ is by all means equivalent to $\lambda z.x$—-this equivalence will be formalized shortly—while $\lambda y.yy$ is not equivalent to $\lambda z.yy$). There are at least three approaches in the literature to deal with this delicate issue:

1. Define $(\lambda y.E)[E'/x]$ as $\lambda y.(E[E'/x])$, but pay special attention whenever substitution is used to add sufficient conditions to assure that y is not free in E'. This approach simplifies the definition of substitution, but complicates the presentation of λ-calculus by having to mention obvious additional hypotheses all the time a substitution is invoked;

2. Define substitution as a *partial operation*: $(\lambda y.E)[E'/x]$ is defined and equal to $\lambda y.(E[E'/x])$ if and only if $y \notin FV(E')$. This may seem like the right approach, but unfortunately is also problematic, because the entire equational definition of λ-calculus would then become *partial*, which has serious technical implications w.r.t. mechanizing equational deduction (or the process of proving λ-expressions equivalent) and rewriting (or reduction);

3. Define substitution as a *total operation*, but apply a renaming of y in $\lambda y.E$ to some variable that does not occur in E or E' in case $y \in FV(E')$ (this renaming is called *α-conversion* and will be defined formally shortly). This approach slightly complicates the definition of substitution, but simplifies the presentation of many results later on. It is also useful when one wants to mechanize λ-calculus, because it provides an algorithmic way to avoid variable capturing:

$$(\lambda y.E)[E'/x] = \begin{cases} \lambda y.(E[E'/x]) & \text{if} \quad y \notin FV(E') \\ \lambda z.((E[z/y])[E'/x]) & \text{if} \quad y \in FV(E') \end{cases}$$

   where z is a new variable that does not occur in E or E'. Note that the the requirement "z does not occur in E or E'" is stronger than necessary, but easier to state that way.

All three approaches above have their advantages and disadvantages, with no absolute best approach. Sections 4.5.9 and 4.5.10 discuss alternative approaches to define λ-calculus, where the explicit lambda bindings and the implicit variable capturing problem are eliminated by design.

4.5.4 Alpha Conversion

In mathematics, functions that differ only in the name of their variables are equal. For example, the functions f and g defined (on the same domain) as $f(x) = x$ and $g(y) = y$ are considered identical. However, with the machinery developed so far, there is no way to show that the λ-expressions $\lambda x.x$ and $\lambda y.y$ are equal. It is common in the development of mathematical theories to add desirable but unprovable properties as axioms. The following is the first meaningful equational axiom of λ-calculus, known under the name of *α-conversion*:

$$\lambda x.E = \lambda z.(E[z/x]), \text{ for any variable } z \text{ that does not occur in } E \qquad (\alpha)$$

The requirement on z not occurring in E is again stronger than necessary, but it is easier to state.

Using the equation above, one has now the possibility to formally prove λ-expressions equivalent. To capture this provability relation formally, we let $E \equiv_\alpha E'$ denote the fact that the equation $E = E'$ can be proved using standard equational deduction from the equational axioms above ((α) plus those for substitution).

4.5.5 Beta Equivalence and Beta Reduction

We now define another important equation of λ-calculus, known under the name of *β-equivalence*:

$$(\lambda x.E)E' = E[E'/x] \qquad (\beta)$$

The equation (β) tells us how λ-abstractions are *applied*. Essentially, the argument λ-expression passed to a λ-abstraction is copied at every free occurrence of the variable bound by the λ-abstraction within its scope.

We let $E \equiv_\beta E'$ denote the fact that the equation $E = E'$ can be proved using standard equational deduction from the equational axioms above: (α), (β), plus those for substitution. For example $(\lambda f.fx)(\lambda y.y) \equiv_\beta x$, because one can first deduce that $(\lambda f.fx)(\lambda y.y) = (\lambda y.y)x$ by (β) and then that $(\lambda y.y)x = x$ also by (β); the rest follows by the transitivity rule of equational deduction.

When the equation (β) is applied only from left to write, that is, as a rewrite rule, it is called *β-reduction*. We let $\Rightarrow_\beta$ denote the corresponding rewriting relation on λ-expressions. To be more precise, the relation $\Rightarrow_\beta$ is defined on α-equivalence classes of λ-expressions; in other words, $\Rightarrow_\beta$ applies *modulo α-equivalence*.

Given a λ-expression E, one can always apply α-conversion on E to rename its binding variables so that all these variables have different names and they do not occur in $FV(E)$. If that is the case, then note that variable capturing cannot occur when applying a β-reduction step. In particular, that means that one can follow the first, i.e., the simplest approach of the three discussed previously in Section 4.5.3 to define or implement substitution. In other words, if one renames the binding variables each time before applying a β-reduction, then one does not need to rename binding variables during substitution. This is so convenient in the theoretical developments of λ-calculus, that most works on this subject adopt the following convention: *all the binding variables occurring in any given λ-expression at any given moment are assumed to be different; moreover, it is assumed that a variable cannot occur both free and bound in any λ-expression.* If a λ-expression does not satisfy this, then one can apply a certain number of α-conversions and eventually transform it into an α-equivalent one that does satisfy it. Clearly, this process of renaming potentially all the binding variables before applying any β-reduction step may be computationally quite expensive. In a more familiar setting, it is like traversing and changing the names of all the variables in a program at each execution step! There are techniques aiming at minimizing the amount of work to be performed in order to avoid variable capturing. All these techniques, however, incur certain overheads.

One should not get tricked by thinking that one renaming of the binding variables, at the beginning of the reduction process, should be sufficient. It is sufficient for just one step of β-reduction, but not for more. Consider, for example, the closed λ-expression, or the combinator, $(\lambda z.zz)(\lambda x.\lambda y.xy)$. It has three binding variables, all distinct, so their renaming makes no difference. However, if one applies substitution in β-reductions blindly then one quickly ends up capturing the variable y:

$$\underline{(\lambda z.zz)(\lambda x.\lambda y.xy)} \Rightarrow_\beta \underline{(\lambda x.\lambda y.xy)(\lambda x.\lambda y.xy)} \Rightarrow_\beta \lambda y.\underline{(\lambda x.\lambda y.xy)\boxed{y}} \Rightarrow_\beta \lambda y.\lambda y.\boxed{y}y$$

Section 4.5.9 will discuss an ingenious technique to transform λ-calculus into combinatory logic which, surprisingly, eliminates the need for substitutions entirely.

Consider the λ-expression $(\lambda f.(\lambda x.fx)y)g$. There are two different ways to apply β-reduction on it:

1. $(\lambda f.\underline{(\lambda x.fx)y})g \Rightarrow_\beta (\lambda f.fy)g$, and
2. $\underline{(\lambda f.(\lambda x.fx)y)g} \Rightarrow_\beta (\lambda x.gx)y$.

Nevertheless, both resulting λ-expressions above can be further reduced to gy by applying β-reduction. This brings us to one of the most notorious results in λ-calculus ($\Rightarrow_\beta^*$ *is the reflexive and transitive closure of* $\Rightarrow_\beta$):

Theorem 23. *$\Rightarrow_\beta$ is confluent. That means that for any λ-expression E, if $E \Rightarrow_\beta^* E_1$ and $E \Rightarrow_\beta^* E_2$ then there is some λ-expression E' such that $E_1 \Rightarrow_\beta^* E'$ and $E_2 \Rightarrow_\beta^* E'$. All this is, of course, modulo α-conversion.*

The confluence theorem above says that it essentially does not matter how the β-reductions are applied on a given λ-expression. A λ-expression is called a *β-normal form* if no β-reduction can be applied on it.

A λ-expression E is said to *admit a β-normal form* if and only if there is some β-normal form E' such that $E \Rightarrow_\beta^* E'$. The confluence theorem implies that if a λ-expression admits a β-normal form then that β-normal form is *unique modulo α-conversion*.

Note, however, that there are λ-expressions which admit no β-normal form. Consider, for example, the λ-expression $(\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$, known as the *divergent combinator* `omega`. It is easy to see that `omega` $\Rightarrow_\beta$ `omega` and that's the only β-reduction that can apply on `omega`, so it has no β-normal form.

4.5.6 Lambda Calculus as a Programming Language

In contrast to Turing machines, which are rather low-level computational devices, λ-calculus is closer in spirit to higher-level programming languages, particularly to functional programming languages. In this section we show how several common programming language features, including particular language constructs, data-types, and recursion, can be naturally represented as λ-expressions. We say that such language features are *derived* (from λ-calculus), or *syntactic sugar* (for equivalent but harder to read λ-expressions). All these suggest that λ-calculus is rich and intuitive enough to safely be regarded as a programming language itself.

Anonymous Functions and Let Binders

Most functional programming languages, and also some other languages which are not necessarily considered functional, provide *anonymous*, or *nameless functions*, which are functions that are not bound to any particular name. For example, the anonymous successor function is written as `fn x => x+1` in the Standard ML functional language. Anonymous functions can still be applied to arguments, for example `(fn x => x+1) 2` evaluates to 3 in Standard ML, and they can typically be passed as arguments to or returned as results by other functions. Almost all languages supporting anonymous functions have a different syntax for them. Nevertheless, anonymous functions are nothing but λ-abstractions. For example, the successor function above can be regarded as the λ-abstraction λ`x.(x+1)`, where `1` and `+` are either builtins or can be also encoded as λ-expressions (these are discussed in the sequel).

Another common construct in programming languages is the `let` binder, which allows to bind expressions to names and then use those names as replacements for their corresponding expressions. For example,

```
let x1 = E1 and x2 = E2 and...and xn = En in E
```

first binds the expressions `E1`, `E2`, ..., `En` to the names `x1`, `x2`, ..., `xn`, respectively, and then `E` is evaluated. It is not hard to see that this construct is equivalent to the λ-expression (λ`x1`.λ`x2` ... λ`xn`.`E`) `E1 E2` ... `En`.

Currying

Recall from Section 2.1.1 that there is a bijection between $[A \times B \to C]$ and $[A \to [B \to C]]$, where $[X \to Y]$ represents the set of functions of domain X and codomain Y. Indeed, any function $f : A \times B \to C$ can be regarded as a function $g : A \to [B \to C]$, where for any $a \in A$, $g(a)$ is defined as the function $h_a : B \to C$ with $h_a(b) = c$ if and only if $f(a, b) = c$. Similarly, any function $g : A \to [B \to C]$ can be regarded as a function $f : A \times B \to C$, where $f(a, b) = g(a)(b)$.

This observation led to the important concept called *currying*, which allows us to eliminate functions with multiple arguments from the core of a language, replacing them systematically by functions admitting only one argument as above. Thus, functions with multiple arguments are just syntactic sugar.

From now on we may write λ-expressions of the form $\lambda xyz \cdots .E$ as shorthands for their *uncurried versions* $\lambda x.\lambda y.\lambda z. \cdots .E$. With this convention, λ-calculus therefore admits multiple-argument λ-abstractions.

Note, however, that unlike in many familiar languages, curried functions can be applied on fewer arguments. For example, $(\lambda xyz.E)E'$ β-reduces to $\lambda yz.(E[E'/x])$. Also, since λ-application was defined to be left-associative, $(\lambda xyz.E)E_1E_2$ β-reduces to $\lambda z.((E[E_1/x])[E_2/y])$.

Most functional languages today support curried functions. The advantage of currying is that one only needs to focus on defining the meaning or on implementing effectively functions of one argument.

Church Booleans

Booleans are perhaps the simplest data-type that one would like to have in a programming language. λ-calculus so far provides no explicit support for Booleans or conditionals. We next show that λ-calculus provides *implicit* support for Booleans. In other words, the machinery of λ-calculus is powerful enough to simulate Booleans and what one would normally want to do with them in a programming language. What we discuss next is therefore a *methodology* to program with Booleans in λ-calculus.

The idea is to regard a Boolean through a behavioral prism: with a Boolean, one can always choose one of any two objects—if true then the first, if false then the second. In other words, one can identify a Boolean b with a universally quantified conditional "for any `x` and `y`, `if` b `then x else y`". With this behavior of Booleans in mind, one can now relatively easily translate Booleans and Boolean operations in λ-calculus:

`true` := $\lambda xy.x$

`false` := $\lambda xy.y$

`if-then-else` := $\lambda xyz.xyz$

`and` := $\lambda xy.(x\, y$ `false`$)$

Other Boolean operations can be defined in a similar style (see Exercise 198).

This encoding for Booleans is known under the name of *Church Booleans*. We can use β-reduction to show, for example, that `and true false` $\Rightarrow_\beta$ `false`, that is, `and true false` $\equiv_\beta$ `false`. We can show relatively easily that the Church Booleans have all the desired properties of Booleans. Let us, for example, show the associativity of `and`. First, note the obvious equivalences:

`and (and x y) z` $\equiv_\beta$ `x y false z false`

`and x(and y z)` $\equiv_\beta$ `x (y z false) false`

These equivalences tell us that we cannot expect the properties of Booleans to hold for any λ-expressions. Therefore, in order to complete the proof of associativity of `and`, we need to make further assumptions regarding the Booleanity of `x`, `y`, `z`. If `x` is `true`, that is $\lambda xy.x$, then both right-hand-side λ-expressions above reduce to `y z false`. If `x` is `false`, that is $\lambda xy.y$, then the first reduces to `false z false` which further reduces to `false`, while the second reduces to `false` in one step. We can similarly prove the desired properties of all the Boolean operators (Exercise 199), relying only on the generic β-equivalence of λ-calculus.

We may often introduce definitions such as the above for the Church Booleans, using the symbol `:=`. Note that this is not a meta binding constructor on top of λ calculus. It is just a way for us to avoid repeating certain frequent λ-expressions; one can therefore regard them as macros. Anyway, they admit a simple translation into standard λ-calculus, using the usual convention for translating bindings. Therefore, one can regard the λ-expression "`and true false`" as syntactic sugar for

(λ`and`.λ`true`.λ`false`. `and true false`) ((λ`false`.$\lambda xy.\ x\, y$ `false`) ($\lambda xy.y$)) ($\lambda xy.x$) ($\lambda xy.y$)

Pairs

λ-calculus can also naturally encode data-structures of interest in most programming languages. The idea is that λ-abstractions, by their structure, can store useful information. Let us, for example, consider pairs as special cases of records. Like Booleans, pairs can also be regarded behaviorally: a pair is a black-box that can store any two expressions and then allow one to retrieve those through appropriate projections. Formally, we would like to define λ-expressions `pair`, `1st` and `2nd` in such a way that for any other λ-expressions E and E', it is the case that `1st` (`pair` E E') and `2nd` (`pair` E E') are β-equivalent to E and E', respectively. Fortunately, these can be defined quite easily:

$$\texttt{pair} := \lambda xyb.\, bxy,$$

$$\texttt{1st} := \lambda p.\, p\,\texttt{true}, \text{ and}$$

$$\texttt{2nd} := \lambda p.\, p\,\texttt{false}.$$

The idea is therefore that `pair` E E' β-reduces to the λ-expression $\lambda b.\, b\,E\,E'$, which freezes E and E' inside a λ-abstraction, together with a handle, b, which is expected to be a Church Boolean, to unfreeze them later. Indeed, the first projection, `1st`, takes a pair and applies it to `true` hereby unfreezing its first component, while the second projection applies it to `false` to unfreeze its second component.

Church Numerals

Numbers and the usual operations on them can also be defined as λ-expressions. The basic idea is to regard a natural number n as a λ-expression that has the potential to apply a given operation n times on a given starting λ-expression. Therefore, λ-numerals, also called *Church numerals*, take two arguments, *what to do* and *what to start with*, and apply the first as many times as the intended numeral on the second. Intuitively, if the action was *successor* and the starting expression was *zero*, then one would get the usual numerals. Formally, we define numerals as follows:

$$0_\lambda := \lambda sz.\, z$$

$$1_\lambda := \lambda sz.\, s\,z$$

$$2_\lambda := \lambda sz.\, s\,(s\,z)$$

$$3_\lambda := \lambda sz.\, s\,(s\,(s\,z))$$

$$\ldots$$

With this intuition for numerals in mind, we can now easily define a successor operation on numerals:

$$\texttt{succ} := \lambda nsz.\, n\,s\,(s\,z)$$

The above says that for a given numeral n, its successor `succ` n is the numeral that applies the given operation s for n times starting with $s\,z$. There may be several equivalent ways to define the same intended meaning. For example, one can also define the successor operation by applying the operation s only once, but on the expression $n\ s\ z$; therefore, we can define

$$\texttt{succ'} := \lambda nsz.\, s\,(n\,s\,z)$$

One may, of course, want to show that `succ` and `succ'` are equal. However, they are *not* equal as λ-expressions. To see it, one can apply both of them on the λ-expression $\lambda xy.\, x$: one gets after β-reduction $\lambda sz.\, s$ and, respectively, $\lambda sz.\, s\ s$. However, they are equal when applied on Church numerals (see Exercise 200).

One can also define addition as a λ-abstraction, e.g., as follows:

$$\texttt{plus} := \lambda mnsz.\, m\ s\ (n\ s\ z)$$

One of the most natural questions that one can and should ask when one is exposed to a new model of natural numbers, is whether it satisfies the Peano axioms. In our case, this translates to whether the following properties hold for all Church numerals n_λ and m_λ (see Exercise 201):

$$\texttt{plus}\ 0_\lambda\ m_\lambda\ \equiv_\beta\ m_\lambda, \text{ and}$$
$$\texttt{plus}\ (\texttt{succ}\ n_\lambda)\ m_\lambda\ \equiv_\beta\ \texttt{succ}\ (\texttt{plus}\ n_\lambda\ m_\lambda).$$

Church numerals in combination with pairs allow us to define certain recursive behaviors. Let us next define a more interesting function on Church numerals, namely one that calculates Fibonacci numbers. Specifically, we want to define a λ-expression `fibo` with the property that `fibo` n_λ β-reduces to the n^{th} Fibonacci number. Recall that Fibonacci numbers are defined recursively as $f_0 = 0$, $f_1 = 1$, and $f_n = f_{n-1} + f_{n-2}$ for all $n \geq 2$. The idea is to define a two-number window that slides through the sequence of Fibonacci numbers until it reaches the desired number. The window is defined as a pair and the sliding by moving the second element in the pair on the first position and placing the next Fibonacci number as the second element. The shifting operation needs to be applied as many times as the index of the desired Fibonacci number:

$$\texttt{start} := \texttt{pair}\ 0_\lambda\ 1_\lambda,$$
$$\texttt{step} := \lambda p.\, \texttt{pair}\ (\texttt{2nd}\ p)\ (\texttt{plus}\ (\texttt{1st}\ p)\ (\texttt{2nd}\ p)),$$
$$\texttt{fibo} := \lambda n.\, \texttt{1st}\ (n\ \texttt{step}\ \texttt{start}).$$

We will shortly discuss a technique to support recursive definitions of functions in a general way, not only on Church numerals.

Another use of the technique above is in defining the predecessor operation on Church numerals:

$$\texttt{start} := \texttt{pair}\ 0_\lambda\ 0_\lambda,$$
$$\texttt{step} := \lambda p.\, \texttt{pair}\ (\texttt{2nd}p)\ (\texttt{plus}\ 1_\lambda\ (\texttt{2nd}p)),$$
$$\texttt{pred} := \lambda n.\, \texttt{1st}\ (n\ \texttt{step}\ \texttt{start}).$$

Note that $\texttt{pred}\ 0_\lambda\ \equiv_\beta\ 0_\lambda$, which is a slight violation of the usual properties of the predecessor operation on integers. The above definition of predecessor is computationally very inefficient. Unfortunately, there does not seem to be any better way to define this operation on Church numerals.

Subtraction can now be defined easily:

$$\texttt{sub} := \lambda mn.\, n\,\texttt{pred}\,m.$$

Note, again, that negative numbers are collapsed to 0_λ.

Let us next see how relational operators can be defined on Church numerals. These are useful to write many meaningful programs. We first define a helping operation, to test whether a number is zero:

`zero?` := $\lambda n.\, n$ (`and` `false`) `true`.

Now the "less than or equal to" (`leq`), the "larger than or equal to" (`geq`), and the "equal to" (`equal`) can be defined as follows:

`leq` := $\lambda mn.$ `zero?` (`sub` m n),

`geq` := $\lambda mn.$ `zero?` (`sub` n m),

`equal` := $\lambda mn.$ `and` (`leq` m n) (`geq` m n).

Adding Built-ins

As shown above, λ-calculus is powerful enough to encode many data-structures and data-types of interest. However, as it is the case with many other *pure* programming paradigms, in order for λ-calculus to be usable as a reasonably efficient programming language, it needs to provide built-ins comprising efficient implementations for common data-types and operations on them.

We here only discuss the addition of *built-in integers* to λ-calculus; the addition of other data-types is similar. We say that the new λ-calculus obtained this way is *enriched*. Depending on how the underlying substitution operation and how the builtin operations are defined, enriching λ-calculus with builtin integers can take very little effort: syntactically, we may only need to add integers as λ-expressions, and semantically, we may only need to add the rules defining the builtin operations to the language, in addition to the already existing β-reduction rule. Specifically, assume a substitution operation that works over any syntactic extension of λ-calculus, essentially applying homomorphically through all non-binding language constructs, like in Exercise 196. Also, assume a syntactic category *Int* for integer values, e.g., 5, 7, etc., and operations on them, e.g., +, together with a rewrite relation capable of reducing *Int* expressions, e.g., 5+7 $\Rightarrow$ 12.

There are at least two ways to enrich λ-calculus with integers: the simplest is to collapse the *Int* and *Exp* syntactic categories; another is to make *Int* a syntactic subcategory of *Exp*, but then all the *Int* operations need to be overloaded to take and yield *Exp*. In both cases, the integer values become particular expressions, and the integer reduction rules harmoniously co-exist with β-reduction. For example, 5+7 still reduces to 12. But what is more interesting is that $\lambda x.$7+5 also makes sense and reduces to $\lambda x.$12 (without applying any β-reduction step, but only the reduction that *Int* provides). Moreover, ($\lambda yx.$7+y) 5 first β-reduces to $\lambda x.$7+5 and then *Int*-reduces to $\lambda x.$12. The above β-reduction works because integer values are regarded as constants, that is, operations with no arguments, so the generic substitution (homomorphically) applies on them, too, keeping them unchanged, as expected: $I[E'/x] = I$, for any integer I.

Note that one can now write λ-expressions that are not well formed, such as the λ application of one integer to another: 7 5. It would be the task of a type checker to catch such kind of errors. We here focus only on the evaluation, or reduction, mechanism of the enriched calculus, assuming that programs are well-formed. Ill-formed λ-expressions may get stuck when β-reduced.

4.5.7 Fixed-Points and Recursion

Recursion almost always turns out to be a subtle topic in foundational approaches to programming languages. We have already seen the divergent combinator

`omega` := ($\lambda x.\, x\ x$) ($\lambda x.\, x\ x$),

which has the property that `omega` $\Rightarrow_\beta$ `omega`, that is, it leads to an infinite recursion. While `omega` has a recursive behavior, it does not give us a principled way to define recursion in λ-calculus.

But what is *recursion*? Or to be more precise, what is a *recursive function*? Let us examine the definition of a factorial function, in some conventional programming language, that we would like to be recursive:

```
function f(x) {
   if x == 0 then 1 else x * f(x - 1)
}
```

In a functional language that is closer in spirit to λ-calculus the definition of factorial would be:

```
letrec
   f(x) = if x == 0 then 1 else x * f(x - 1)
in f(3) .
```

Note that the `letrec` binding is necessary in the above definition. If we used `let` instead, then according to the syntactic sugar transformation of functional bindings into λ-calculus, the above would be equivalent to

```
(λ f . f 3)
 (λ x . if x == 0 then 1 else x * f(x - 1)) ,
```

so the underlined `f` is free rather than bound to λ `f`, as expected. A functional programming language would report an error here.

The foundational question regarding recursion in λ-calculus is therefore the following: how can we define a λ-abstraction

```
f := <begin-exp ... f ... end-exp>,
```

that is, one in which the λ-expression refers to itself in its scope? Let us put the problem in a different light. Consider instead the well-formed well-behaved λ-expression

```
F := λ f .  <begin-exp ... f ... end-exp>,
```

that is, one which takes any λ-expression, in particular a λ-abstraction, and plugs it at the right place into the scope of the λ-expression that we want to define recursively. The question now becomes the following one: can we find a *fixed point* `f` of `F`, that is, a λ-expression `f` with the property that

$$\texttt{F f} \equiv_\beta \texttt{f}\,?$$

Interestingly, λ-calculus has the following notorious and perhaps surprising result:

Theorem 24. ***(Fixed-point theorem of untyped λ-calculus)*** *For any λ-expression F, there is some λ-expression X such that $FX \equiv_\beta X$.*

Proof. One such X is the λ-expression $(\lambda x.\, F\,(x\,x))\,(\lambda x.\, F\,(x\,x))$. Indeed,

$$\begin{array}{lll} X & = & (\lambda x.\, F\,(x\,x))\,(\lambda x.\, F\,(x\,x)) \\ & \equiv_\beta & F\,((\lambda x.\, F\,(x\,x))\,(\lambda x.\, F\,(x\,x))) \\ & = & F\,X. \end{array}$$

Therefore, X is a fixed-point of F. □

The proof of the fixed-point theorem above suggests defining the following famous *fixed-point combinator*:

$$Y \;:=\; \lambda F.\,(\lambda x.\, F\,(x\,x))\,(\lambda x.\, F\,(x\,x)).$$

With this, for any λ-expression F, the λ-application $Y\,F$ becomes the fix-point of F; therefore, $F\,(Y\,F) \equiv_\beta Y\,F$. Thus, we have a constructive way to build fixed-points for any λ-expression F. Note that F does not need to be a λ-abstraction.

Let us now return to the recursive definition of factorial in λ-calculus enriched with integers. For this particular definition, let us define the λ-expression:

```
F := λ f . λ x . (if x == 0 then 1 else x * f(x - 1))
```

The recursive definition of factorial is therefore a fixed-point of `F`, namely, Y`F`. It is such a fixed-point λ-expression that the `letrec` functional language construct in the definition of factorial refers to!

Let us experiment with this λ-calculus definition of factorial, by calculating factorial of 3:

```
(Y F) 3 ≡β
F (Y F) 3 =
(λ f . λ x . (if x == 0 then 1 else x * f(x - 1))) (Y F) 3 ⇒β
if 3 == 0 then 1 else 3 * (Y F)(3 - 1) ⇒
3 * ((Y F) 2) ≡β
...
6 * ((Y F) 0) ≡β
6 * (F (Y F) 0) =
6 * ((λ f . λ x . (if x == 0 then 1 else x * f(x - 1))) (Y F) 0) ⇒β
6 * if 0 == 0 then 1 else 0 * (Y F)(0 - 1) ⇒
6 * 1 ⇒
6
```

Therefore, λ-calculus can be regarded as a simple programming language, providing support for functions, numbers, data-structures, and recursion. It can be shown that any computable function can be expressed in λ-calculus in such a way that its computation can be performed by β-reduction. This means that λ-calculus is a *Turing-complete model of computation*.

There are two aspects of λ-calculus that lead to complications when one wants to implement it.

One is, of course, the substitution: efficiency and correctness are two opposing tensions that one needs to address in any direct implementation of λ-calculus.

The other relates to the strategies of applying β-reductions: so far we used what is called *full β-reduction*, but other strategies include *normal evaluation*, *call-by-name*, *call-by-value*, etc. There are λ-expressions whose β-reduction does not terminate under one strategy but terminates under another. Moreover, depending upon the strategy of evaluation employed, other fixed-point combinators may be more appropriate.

Like β-reduction, the evaluation of expressions is confluent in many pure functional languages. However, once a language allows side effects, strategies of evaluation start playing a crucial role; to avoid any confusion, most programming languages hardwire a particular evaluation strategy, most frequently *call-by-value*.

We do not discuss strategies of evaluation here. Instead, we approach the other delicate operational aspect of λ-calculus, namely the substitution. In fact, we show that it can be completely eliminated if one applies a systematic transformation of λ-expressions into expressions over a reduced set of combinators.

More precisely, we show that any closed λ-expression can be systematically transformed into a λ-expression build over only the combinators $K := \lambda xy.x$ and $S := \lambda xyz.xz(yz)$, together with the λ-application operator. For example, the identity λ-abstraction $\lambda x.x$ is going to be $S\,K\,K$; indeed,

$S\,K\,K \equiv_\beta \lambda z.Kz(Kz) = \lambda z.(\lambda xy.x)z(Kz) \equiv_\beta \lambda z.z \equiv_\alpha \lambda x.x$.

Interestingly, once such a transformation is applied, one will not need the machinery of λ-calculus and β-reduction anymore. All we'll need to do is to capture the contextual behavior of K and S, which can be defined equationally very elegantly: $KXY = X$ and $SXYZ = XZ(YZ)$, for any other KS-expressions X, Y, Z.

Before we do that, we need to first discuss two other important aspects of λ-calculus: η-equivalence and extensionality.

4.5.8 Eta Equivalence and Extensionality

Here we discuss two common but equivalent extensions of λ-calculus, both concerned with additional proof support for equality of λ-expressions (besides α-conversion and β-equivalence). The former is called *η-equivalence* and allows us to state that expressions $\lambda x.\, E\, x$ and E are always equivalent, provided that x does not occur free in E. The latter is called *extensionality* and allows us to derive that E and E' are equal provided that $E\, x$ and $E'\, x$ are equal for some variable x that does not occur free in any of E and E'.

Eta Equivalence

Let us consider the λ-expression $\lambda x.Ex$, where E is some λ-expression that does not contain x free. Intuitively, $\lambda x.Ex$ does nothing but wraps E: when called, it passes its argument to E and then passes back E's result. When applied on some λ-expression, say E', note that $\lambda x.Ex$ and E *behave the same*. Indeed, since E does not contain any free occurrence of x, one can show that $(\lambda x.Ex)E' \equiv_\beta EE'$. Moreover, if E is a λ-abstraction, say $\lambda y.F$, then $\lambda x.Ex = \lambda x.(\lambda y.F)x \equiv_\beta \lambda x.F[x/y]$. The latter is α-equivalent to $\lambda y.F$, so it follows that in this case $\lambda x.Ex$ is β-equivalent to E.

Even though $\lambda x.Ex$ and E have similar behaviors in applicational contexts and they can even be shown β-equivalent when E is a λ-abstraction, there is nothing to allow us to use their equality as an axiom in our equational inferences. In particular, there is no way to show that the combinator $\lambda x.\lambda y.xy$ is equivalent to $\lambda x.x$. To increase the proving capability of λ-calculus, still without jeopardizing its basic intuitions and applications, we consider its extension with the following equation:

$$(\eta) \qquad \lambda x.Ex = E\,,$$

for any $x \notin FV(E)$. We let $E \equiv_{\beta\eta} E'$ denote the fact that the equation $E = E'$ can be proved using standard equational deduction from all the equational axioms above: (α), (β), (η), plus those for substitution. The relation $\equiv_{\beta\eta}$ is also called *$\beta\eta$-equivalence*. The λ-calculus enriched with the rule (η) is also written $\lambda + \eta$.

Extensionality

Extensionality is a deduction rule encountered in several branches of mathematics and computer science. It intuitively says that in order to prove two objects equal, one may first extend them in some rigorous way. The effectiveness of extensionality comes from the fact that it may often be the case that the extended versions of the two objects are easier to prove equivalent.

Extensionality was probably first considered as a proof principle in set theory. In naive set theory, sets are built in a similar fashion to Peano numbers, that is, using some simple constructors (together with several constraints), such as the empty set $\emptyset$ and the list constructor $\{x_1, \ldots, x_n\}$. Thus, $\{\emptyset, \{\emptyset, \{\emptyset\}\}\}$ is a well-formed set. With this way of constructing sets, there may be the case that two sets with the same elements have totally different representations. Consequently, it is almost impossible to prove any meaningful property on sets, such as distributivity of union and intersection, etc., by just taking into account how sets are constructed. In particular, proofs by structural induction are close to useless.

Extensionality is often listed as the first axiom in any axiomatization of set theory. In that context, it basically says that two sets are equal iff they have the same elements. Formally, sets S and S' are equal whenever we can prove the statement "for any element x, $x \in S$ iff $x \in S'$". Therefore, in order to show sets S and S' equal, we can first extend them (regarded as syntactic terms) by applying them the membership operator. In most cases the new task is easier to prove.

Similarly, the principle of extensionality applies to functions: two functions are equal whenever they evaluate to the same values for the same elements. Formally, functions f and g are equal whenever we can prove the statement "for any element x, $f(x) = g(x)$". We again have to prove a syntactically extended task, but like for sets it is often the case that the extended task is simpler to prove.

In λ-calculus, extensionality takes the following shape:

(ext) `If` $Ex = E'x$ `for some` $x \notin FV(EE')$, `then` $E = E'$.

Therefore, two λ-abstractions are equal if they are equal when applied on some variable that does not occur free in any of them. Note that "for some x" can be replaced by "for any x" in `ext`. We let $E \equiv_{\beta\texttt{ext}} E'$ denote the fact that the equation $E = E'$ can be proved using standard equational deduction using (α) and (β), together with `ext`. The λ-calculus extended with `ext` is also called λ + `ext`.

The following important result says the extensions of λ-calculus with (η) and with `ext` are equivalent:

Theorem 25. *$\lambda + \eta$ is equivalent to λ + `ext`.*

Proof. In order to show that two mathematical theories are equivalent, one needs to show two things: (1) how the syntax of one translates into the syntax of the other, or in other words to show how one can mechanically translate assertions in one into assertions in the other, and (2) that all the axioms of each of the two theories can be proved from the axioms of the other, along the corresponding translation of syntax. In our particular case of $\lambda + \eta$ and λ + `ext`, syntax remains unchanged when moving from one logic to another, so (1) above is straightforward. We will shortly see another equivalence of logics, where (1) is rather involved. Regarding (2), all we need to show is that under the usual λ-calculus with (α) and (β), the equation (η) and the principle of extensionality are equivalent.

Let us first show that (η) implies `ext`. For that, let us assume that $Ex \equiv_{\beta\eta} E'x$ for some λ-expressions E and E' and for some variable $x \notin FV(EE')$. We need to show that $E \equiv_{\beta\eta} E'$:

$$E \equiv_{\beta\eta} \lambda x.Ex \equiv_{\beta\eta} \lambda x.E'x \equiv_{\beta\eta} E'.$$

Note the use of $\equiv_{\beta\eta}$ in the equivalences above, rather than just $\equiv_\beta$. That is because, in order to prove the axioms of the target theory, λ + `ext` in our case, one can use the entire calculus machinery available available in the source theory, $\lambda + \eta$ in our case.

Let us now prove the other implication, namely that `ext` implies (η). We need to prove that $\lambda x.Ex \equiv_{\beta\texttt{ext}} E$ for any λ-expression E and any $x \notin FV(E)$. By extensionality, it suffices to show that $(\lambda x.Ex)x \equiv_{\beta\texttt{ext}} Ex$, which follows immediately by β-equivalence because x is not free in E. □

4.5.9 Combinatory Logic

Even though λ-calculus can be defined equationally and is a relatively intuitive framework, as we have noticed several times by now, substitution makes it non-trivial to implement effectively. There are several approaches in the literature addressing the subtle problem of automating substitution to avoid variable capture, all with their advantages and disadvanteges. We here take a different approach. We show how λ-expressions can be

automatically translated into expressions over combinators, in such a way that substitution will not even be needed anymore.

A question addressed by many researchers several decades ago, still interesting today and investigated by many, is whether there is any *simple* equational theory that is entirely equivalent to λ-calculus. Since λ-calculus is Turing complete, such a simple theory may provide a strong foundation for computing.

Combinatory logic was invented by Moses Shönfinkel in 1920. The work was published in 1924 in a paper entitled *"On the building blocks of mathematical logic"*. Combinatory logic is a simple equational theory over two sorts, *Var* and *Exp* with $Var < Exp$, a potentially infinite set x, y, etc., of constants of sort *Var* written using lower-case letters, two constants K and S of sort *Exp*, one application operation with the same syntax and left-associativity parsing convention as in λ-calculus, together with the two equations

$$KXY = X,$$
$$SXYZ = XZ(YZ),$$

quantified universally over X, Y, Z of sort *Exp*. The constants K and S are defined equationally in such a way to capture the intuition that they denote the combinators $\lambda xy.x$ and $\lambda xyz.xz(yz)$, respectively. The terms of the language, each of which denoting a function, are formed from variables and constants K and S by a single construction, function application. For example, $S(SxKS)yS(SKxK)z$ is a well-formed term in combinatory logic, denoting some function of free variables x, y, and z.

Let CL be the equational theory of combinatory logic above. Note that a function FV returning the free variables that occur in a term in combinatory logic can be defined in a trivial manner, because there are no bound variables in CL. Also, note that the extensionality principle from λ-calculus translates unchanged to CL:

(ext) If $Ex = E'x$ for some $x \notin FV(EE')$, then $E = E'$.

Let CL + ext be CL enriched with the principle of extensionality. The following is a landmark result:

Theorem 26. *λ + ext is equivalent to CL + ext.*

Proof. Let us recall what one needs to show in order for two mathematical theories to be equivalent: (1) how the syntax of one translates into the syntax of the other; and (2) that all the axioms of each of the two theories can be proved from the axioms of the other, along the corresponding translation of syntax.

Let us consider first the easy part: λ + ext implies CL + ext. We first need to show how the syntax of CL + ext translates into that of λ + ext. This is easy and it was already mentioned before: let K be the combinator $\lambda xy.x$ and let S be the combinator $\lambda xyz.xz(yz)$. We then need to show that the two equational axioms of CL + ext hold under this translation: they can be immediately proved by β-equivalence. We also need to show that the extensionality in CL + ext holds under the above translation: this is obvious, because it is exactly the same as the extensionality in λ + ext.

Let us now consider the other, more difficult, implication. So we start with CL + ext, where K and S have no particular meaning in λ-calculus, and we need to define some map that takes any λ-expression and translates it into an expression in CL.

To perform such a transformation, let us add syntax for λ-abstractions to CL, but without any of the equations of λ-calculus. This way one can write and parse λ-expressions, but still have no meaning for those. The following ingenious *bracket abstraction* rewriting system transforms any uninterpreted λ-expression into an expression using only K, S, and the free variables of the original λ-expression:

1. $\lambda x.\rho \Rightarrow [x]\rho$
2. $[x]y \Rightarrow \begin{cases} SKK & \text{if } x = y \\ Ky & \text{if } x \neq y \end{cases}$
3. $[x](\rho\rho') \Rightarrow S([x]\rho)([x]\rho')$
4. $[x]K \Rightarrow KK$
5. $[x]S \Rightarrow KS$

The first rule removes all the λ-bindings, replacing them by corresponding bracket expressions. Here ρ and ρ' can be any expressions over K, S, variables, and the application operator, but also over the λ-abstraction operator $\lambda\_.\_ : Var \to Exp$. However, note that rules 2-5 systematically elliminate all the brackets. Therefore, the *bracket abstraction* rules above eventually transform any λ-expression into an expression over only K, S, variables, and the application operator.

The correctness of the translation of λ `+ ext` into `CL + ext` via the bracket abstraction technique is rather technical: one needs to show that the translated versions of equations in λ can be proved (by structureal induction) using the machinery of `CL + ext`.

Exercise 185. ***(Technical)*** *Prove the correctness of the translation of λ `+ ext` into `CL + ext` above.*

We do not need to understand the details of the proof of correctness in the exercise above in order to have a good intuition on why the bracket abstraction translation works. To see that, just think of the bracket abstraction as a means to associate equivalent λ-expressions to other λ-abstractions, within the framework of λ-calculus, where K and S are their corresponding λ-expressions. As seen above, it eventually reduces any λ-expression to one over only combinators and variables, containing no explicit λ-abstractions except those that define the combinators K and S. To see that the bracket abstraction is correct, we can think of each bracket term $[x]E$ as the λ-expression that it was generated from, $\lambda x.E$, and then show that each rule in the bracket abstraction transformation is sound within λ-calculus. For example, rule 3 can be shown by extensionality: $(\lambda x.\rho\rho')z \equiv_\beta (\rho[z/x])(\rho'[z/x]) \equiv_\beta ((\lambda x.\rho)z)((\lambda x.\rho')z)) \equiv_\beta (\lambda xyz.xz(yz))(\lambda x.\rho)(\lambda x.\rho')z = S(\lambda x.\rho)(\lambda x.\rho')z$, so by extensionality, $\lambda x.\rho\rho' \equiv_{\beta\text{ext}} S(\lambda x.\rho)(\lambda x.\rho')$.

This way, one can prove the soundness of each of the rules in the bracket abstraction translation. As one may expect, the tricky part is to show the completeness of the translation, that is, that everything one can do with λ-claculus and `ext` can also do with with its subcalculus `CL + ext`. This is not hard, but rather technical.

Exercise 186. *Define the bracket abstraction translation above formally in Maude. To do it, first define `CL`, then add syntax for λ-abstraction and bracket to `CL`, and then add the bracket abstraction rules as equations (which are interpreted as rewrite rules by Maude). Convince yourself that substitution is not a problem in `CL`, by giving an example of a λ-expression which would not be reducible with the definition of λ-calculus in Exercise* **??***, but whose translation in `CL` can be reduced with the two equations in `CL`.*

---

some additional notes:

In the pure system there is a small set of constants all of which are themselves pure combinators, usually the primitive combinators are named S, K and I, though I is in fact definable in terms of S and K.

The intended meaning of these constants can be described simply by showing a corresponding transformation or reduction which the constant may be thought of as effecting.

The transformations are as follows: I x = x K x y = x S f g x = (f x)(g x) This set of combinators has the property of combinatory completeness which means that for any function definition of the form: f x = combinatory expression involving x

There exists an expression E in S,K and I such that: f = E

This property establishes the close relationship between Pure Combinatory Logic and the Pure lambda Calculus, in which a special notation for functional abstraction is available. It shows that the notation for functional abstraction, though a great convenience, adds no expressiveness to the language.

As well as having combinatorial completeness, Pure Combinatory Logic is able to express all effectively computable functions over natural numbers appropriately represented as combinators.

[Barenderegt84] Barendregt, H.P.; The Lambda Calculus - Its Syntax and Semantics; Second Edition, North Holland 1984 purchase from amazon purchase from ibs

This is the first published work in the field which came later to be known as combinatory logic.

Schonfinkel shows how the use of bound variables in logic can be dispensed with. The use of higher order functions makes possible the reduction of logic to a language consisting of one constructor (the application of a function to an argument) and three primitive constants U, C (now usually called K) and S. A function is termed ”higher order” if it will accept a function as an argument, or return one as its result. U, C, and S are all higher order functions.

The meaning of these primitives may be understood through the following informal definitions:

Cxy = (C(x))(y) = x C is a function which given any value x, returns the constant x valued function. Sfgx = ((S(f))(g))(x) = (f(x))(g(x)) S is a function which combines two functions, say f and g, supplied as successive arguments. The resulting function, given a value x, returns the value obtained by applying the value of f at x to the value of g at x. UPQ = (U(P))(Q) = forall x. not(P(x) and Q(x)) U is a generalised scheffer stroke. It should be thought of as applying to two predicates and returns the universal quantification of the negated conjunction of the two predicates.

These combinators are sufficient to enable arbitrary first order predicates to be expressed without the use of bound variables, which appear in first order logic whenever a quantifier is used. This can be demonstrated most straightforwardly using a simple algorithm which converts lambda-expressions to combinators. They are not limited to first-order predicates, but without some constraints (equivalent to those found in first order logic, or more liberally and appropriately to those in Church’s Simple Theory of Types [Church40]) the logic which results from the use of these combinators is at risk of proving inconsistent. Combinatory logic has traditionally tried to avoid type constraints and has therefore been dogged with difficulties in achieving strength and consistency.

Schonfinkel’s paper remains an accessible introduction to combinatory logic which makes clear the original motivation for this innovation. This original motivation was vigorously pursued later by H.B.Curry and his collaborators. The results of Curry’s programme were published in Combinatory Logic, Vols 1 & 2.

While Schofinkel was successful in showing that combinators could provide plausible notation, the semantic and proof theoretic aspects were more difficult to put in place. The difficulty is more pronounced if it hoped to use combinatory logic as a foundation for mathematics, than it is for the use of combinators in less demanding roles (e.g. for programming languages).

The Pure combinatory logic may be thought of as Schonfinkels system with the logical operator ”U” omitted (though the presentation may vary). Illative combinatory logics are formed by adding additional logical primitives to the pure combinators. There are many illative combinatory logics, which vary among themselves in substance as well as in presentation.

Pure combinatory logic is so closely related to Church's lambda-calculus that it is best studied alongside the lambda-calculus, for which the most comprehensive modern text is probably The Lambda Calculus. A very popular and entertaining introduction to the pure combinators may be found in To Mock a Mockingbird.

Research on illative combinatory logics has been continued by Curry's students, Roger Hindley and Jonathan Seldin, and by Martin Bunder. Hindley, Lercher and Seldin published An Introduction to Combinatory Logic, an excellent and readable short introduction to the subject, now superseded by the more comprehensive An Introduction to Combinators and the Lambda-Calculus

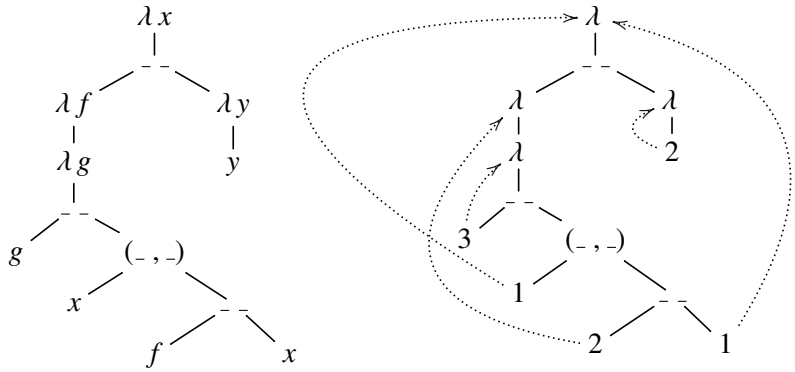
Recently Randall Holmes has developed logical systems based on combinatory logic and the lambda calculus using methods derived from Quine's formulations of set theory.

The attempt to make a foundation for mathematics using the illative combinatory logic were blighted by Curry's paradox and semantic opacity. While systems avoiding this paradox have been devised, they are typically weak, and lack the kind of convincing semantic intuition found in first order set theory.

[Curry58] Curry, Haskell B., Feys, Robert; Combinatory Logic, Volume I; North Holland 1958; [Curry72] Curry, Haskell B., Hindley, J.Roger, Seldin, Jonathan P.; Combinatory Logic, Volume II; North Holland 1972; ISBN 0 7204 2208 6

4.5.10 De Bruijn Nameless Representation

Interestingly, note that the morphism obtained above contains no references to the variables that occur in the original λ-expression. It can be shown that the interpretation of λ-expressions into a CCC is *invariant to α-conversion*. To see that, let us draw the morphism above as a tree, where we write λ instead of *curry*, $\_\ \_$ instead of $\langle \_ , \_ \rangle$; $app^{[\![s]\!],[\![t]\!]}$, $(\_ , \_)$ instead of the remaining $\langle \_ , \_ \rangle$ and i instead of π_i (and omit the types):



The (right) tree above suggests a representation of λ-expressions that is invariant to α-conversion: each binding variable is replaced by a natural number, representing the number of λs occurring on the path to it; that number then replaces consistently all the bound occurrences of the variable. The corresponding lambda expression without variables obtained using this transformation is $\lambda\,(\lambda\,\lambda\,(3\ (1, (2\ 1)))\ \lambda\,2)$.

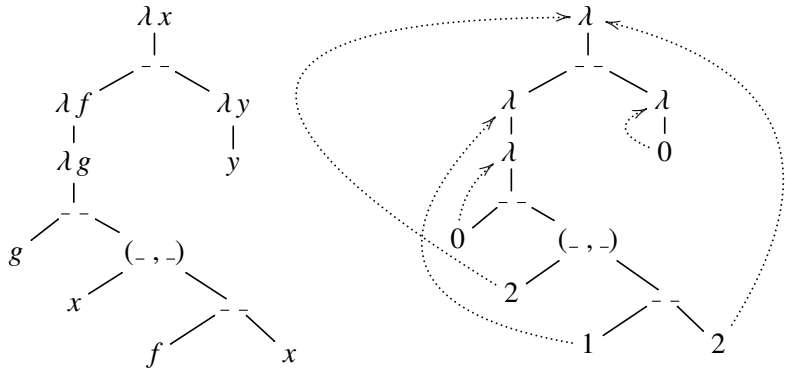
Exercise 187. *Explain why this representation is invariant to α-conversion.*

The representation of α-expressions above was explicitly proposed as a means to implement λ-calculus in 1971 by Nicholas de Bruijn.

In the same paper, de Bruijn proposed another encoding which became more popular. We do not know whether de Bruijn was influenced by the CCC interpretation of λ-expressions or not, but we discuss his other representation technique here.

de Bruijn Nameless Representation of λ-expression

The second and more popular representation technique of λ-expressions proposed by Nicholas de Bruijn in 1971 is a bottom-up version of the above representation. For the above example the tree representing the encoding is (we omit the types):



In this encoding, each variable is replaced by the number of lambda abstractions on the path from it to the lambda abstraction binding it. The encoding for the given example is $\lambda\,(\lambda\,\lambda\,(0\ (2, (1\ 2)))\ \lambda\,0)$.

One can easily define application for the above de Bruijn encoding:

$(\lambda\ E\ E') \Rightarrow E[E'/0]$
$(E\ E')[E''/i] \Rightarrow (E[E''/i])\ (E'[E''/i])$ $(\lambda\ E)[E'/i] \Rightarrow \lambda\,(E[\uparrow\ E'/(i1 +_{Int})])$ $j[E/i] \Rightarrow$ if $j == i$ then E else (if $j > i$ then $j - 1$ else j fi) fi
$\uparrow\ E' \Rightarrow \uparrow^0 E$
$\uparrow^i\ (E\ E') \Rightarrow (\uparrow^i E)\ (\uparrow^i E')$ $\uparrow^i\ (\lambda\ E) \Rightarrow \lambda\,(\uparrow^{i+1} E)$ $\uparrow^i\ j \Rightarrow$ if $j >= i$ then $(j + 1)$ else j fi

Exercise 188. *Define the transformation above formally in Maude.*

Exercise 189. *Define application for the first de Bruijn encoding, in a similar style to the one above.*

4.5.11 Notes

4.5.12 Exercises

Exercise 190. ★ *Define the syntax of λ-calculus in a* `Maude` *module using mix-fix notation. Then parse the various λ-expressions that were discussed in this section.*

Exercise 191. ★ *Extend the Maude definition of λ-calculus in Exercise 190 with a definition of free variables. You should define an operation* `fv` *taking an expression and returning a set of variables.*

Exercise 192. *Prove the following equivalences of λ-expressions:*

- $\lambda x.x \equiv_\alpha \lambda y.y$,
- $\lambda x.x(\lambda y.y) \equiv_\alpha \lambda y.y(\lambda x.x)$,
- $\lambda x.x(\lambda y.y) \equiv_\alpha \lambda y.y(\lambda y.y)$.

Exercise 193. *Show that* $(\lambda x.(\lambda y.x))yx \equiv_\beta y$.

Exercise 194. ★ *Define substitution in Maude, as a partial operation. As discussed in Section 4.5.3, a partial substitution applies only if it does not lead to a variable capture; otherwise the substitution operation is undefined. In other words, when applying a partial substitution we do not need to perform any α-conversions.*
Hint. *Define* $(\lambda y.E)[E'/x]$ *when* $y \neq x$ *as a conditional equation, with condition* $y \notin FV(E')$.

Exercise 195. ★ *Define unrestricted substitution in Maude, which takes into account and avoids variable capture. Use the core-level Maude capabilities for this exercise (see also Exercise 196).*
Hint. *You need a mechanism to generate fresh variables. For example, you may define an operation* `var` *from Int to Var, where* `var`(1), `var`(2)*, etc., represent fresh variables. A simple but wasteful solution when calculating* $(\lambda y.E)[E'/x]$ *is to always replace* y *in* $\lambda y.E$ *by a fresh variable before applying the substitution; a better solution is to only do it when* $y \in FV(E')$.

Exercise⋆ 196. ★ *Define unrestricted substitution in Maude, using the meta-level capabilities of Maude (see also Exercise 195). You should traverse the meta-term and propagate the substitution operation through any operation label which is not a λ-binding. For λ-bindings you should apply the same conceptual operation as in Exercise 195. The advantage of the meta-level approach is that it is more modular that the other approaches. We will be able to add new constructs to λ-calculus, for example builtins, without having to change the definition of the substitution.*

Exercise 197. ★ *Define λ-calculus formally in Maude, using each of the substitution approaches in Exercises 194, 195, and 196. For the former approach (partial substitution), show that there are λ-expressions that cannot be β-reduced automatically with your definition of λ-calculus to a normal form, even though they are closed (or combinators) and all the binding variables are initially distinct from each other.*

Exercise 198. *Define the other Boolean operations which were not already defined in Section 4.5.6 (including at least* `or`*,* `not`*,* `implies`*,* `iff`*, and* `xor`*) as λ-expressions.*

Exercise 199. *Prove that the Church Boolean operators defined in Exercise 198 have all their desired properties (a set of such desired properties can be found in the Maude BOOL module; use the command "*`show module BOOL`*" lists them).*

Exercise 200. *With the notions and notations in Section 4.5.6, show that for any Church numeral* n_λ*, both* `succ` n_λ *and* `succ'` n_λ *represent the same numeral, namely* $(n+1)_\lambda$.
Hint. *Induction on the structure of* n_λ.

Exercise 201. *Prove that Church numerals form indeed a model of natural numbers, by showing the two properties derived from Peano's axioms (see Section 4.5.6).*

Exercise 202. *Define multiplication on Church numerals and prove its Peano properties.*
Hint. *Multiplication can be defined several different interesting ways.*

Exercise 203. *Define the power operator (raising a number to the power of another) using Peano-like axioms. Then define power on Church numerals and show that it satisfies its Peano axioms.*

Exercise 204. *Add builtin integers to λ-calculus in Maude, extending Exercise 197 based on substitution as defined in Exercise 196.*
Hint. *The only change needed should be to subsort* `Int` *to* `Exp`*. Since Maude's parsing is at the level of kinds, instead of sorts, we should not need to overload the* `Int` *operations to take and return* `Exp` *sorts.*