CS522 - Programming Language Semantics

Lambda Calculus and Combinatory Logic

Grigore Roşu

Department of Computer Science

University of Illinois at Urbana-Champaign

In this part of the course we discuss two important and closely related mathematical theories:

- *Lambda calculus*, written also λ-*calculus*, is a pure calculus of functional abstraction and function application, with many applications in logic and computer science;

- *Combinatory logic* shows that bound variables can be entirely eliminated without loss of expressiveness. It has applications both in the foundations of mathematics and in the implementation of functional programming languages.

A good reference for these subjects is the book *"The Lambda Calculus: Its Syntax and Semantics"* by *H.P. Barendregt (Second Edition, North Holland 1984)*. This book also contains a great discussion on the history and motivations of these theories.

Lambda Calculus (λ-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. Even though the issue of whether λ-calculus indeed provides a strong foundation of mathematics is still largely open, it nevertheless turned out to be 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*. That means that its syntax can be defined as an algebraic signature (to enhance readability we can use the mix-fix notation); 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).

We can therefore conclude that equational logic and rewriting also form a strong foundational framework to describe and explain programming language concepts. This hypothesis was practically evaluated through several concrete definitions of languages in CS422, a course on programming language design.

We will later see in this class that equational logic forms indeed a strong foundation for programming language semantics, providing a framework that supports both *denotational* and *operational* semantics in a unified manner. Moreover, *rewriting logic*, which is a natural extension of equational logics with rewrite rules, provides a foundation for *concurrent* programming language semantics.

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 is considered by some computer scientists "too general": it gives one "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.

Personal note: I disagree with the above criticisms on equational logic in particular, and on rewriting logic in general. What these logics need to become a broadly accepted strong foundation for programming languages is, in my personal view, *good methodologies* to define languages (and this is what we are developing at UIUC in several research projects and courses).

Syntax of λ-Calculus

Assume an infinite set of *variables*, or *names*, V. Then the syntax of *λ-expressions* is (in BNF notation)

$$Exp ::= Var \mid Exp\ Exp \mid \lambda Var.Exp$$

where *Var* ranges over the variables in V. 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 $E_1 E_2$ are called *λ-applications*. The former captures the intuition of "function definition", while the latter that of "function application". To avoid parentheses, assume that λ-application is left associative and binds tighter than λ-abstraction.

Exercise 1 *Define the syntax of λ-calculus in a* `Maude` *module using mix-fix notation; then parse some lambda expressions.*

Many programming language concepts, and even entire programming languages, translate relatively naturally into λ-calculus concepts or into λ-expressions. In particular, one can define some transformation φ from functional expressions into λ-expressions. Such a transformation φ would take, for example,

- variable names `x` to unique variables $x \in Var$;

- function declarations of the form `fun x -> E` to $\lambda x.\varphi(\texttt{E})$; and

- bindings (which generalize the idea of "local declarations" occurring in most programming languages, functional or not) `let x1 = E1 and x2 = E2 and ... and xn = En in E` to λ-expressions $(\lambda x_1.\lambda x_2.\cdots.\lambda x_n.\varphi(\texttt{E}))\varphi(\texttt{E1})\varphi(\texttt{E2})\cdots\varphi(\texttt{En})$.

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 λ *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:

- $FV(x) = \{x\}$,

- $FV(E_1 E_2) = FV(E_1) \cup FV(E_2)$, and

- $FV(\lambda x.E) = FV(E) - \{x\}$.

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*.

Exercise 2 *Extend your Maude definition of λ-calculus in the previous exercise with a definition of free variables. You should define an operation fv taking an expression and returning a set of variables (recall how sets are defined in Maude; if you don't remember, ask!).*

Substitution

Evaluation of λ-expressions is "by-substitution". That means that the λ-expression that is "passed" to a λ-abstraction is "copied as it 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, $E[x \leftarrow E']$ represents the λ-expression obtained from E by replacing each free occurrence of x by E'. Formally, substitution can be defined as follows:

- $y[x \leftarrow E'] = \begin{cases} E' & \text{if } y = x \\ y & \text{if } y \neq x, \end{cases}$

- $(E_1 E_2)[x \leftarrow E'] = (E_1[x \leftarrow E'])(E_2[x \leftarrow E']),$

- $(\lambda x.E)[x \leftarrow E'] = \lambda x.E.$

The tricky part is to define substitution on λ-abstractions of the

form $(\lambda y.E)[x \leftarrow E']$, 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 $\lambda y.(E[x \leftarrow E'])$ (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 $(\lambda y.x)[x \leftarrow yy]$; 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 $(\lambda y.E)[x \leftarrow E']$ as $\lambda y.(E[x \leftarrow E'])$, 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*: $(\lambda y.E)[x \leftarrow E']$ is defined and equal to $\lambda y.(E[x \leftarrow E'])$ 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 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:

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

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, and one can find many scientists defending each of them. However, we will later on choose a totally different approach to define λ-calculus as an executable specification, in which substitutions play no role anymore. More precisely, we will define λ-calculus through its translation to combinatory logic.

α-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 a common phenomenon 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*:

$$(\alpha) \qquad \lambda x.E = \lambda z.(E[x \leftarrow z])$$

for any variable z that does not occur in E (this requirement on z is again stronger than necessary, but it is easier to state).

Using the equation above, one has now the possibility to 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).

Exercise 3 *Prove the following equivalences of λ-expressions:*

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

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

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

β-Equivalence and β-Reduction

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

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

The equation (β) tells us how λ-abstractions are "applied". Essentially, it says that the argument λ-expression that is 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.

Exercise 4 *Show that* $(\lambda x.(\lambda y.x))yx \equiv_\beta y$

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 which 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 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 of the works on this subject make 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 the above convention 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 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 captures. 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, e.g., the closed λ-expression, or the combinator,

$$(\lambda z.zz)(\lambda x.\lambda y.xy).$$

It has three binding variables, all different. However, if one applies substitution in β-reductions blindly then one quickly ends up capturing the variable y:

$$(\lambda z.zz)(\lambda x.\lambda y.xy) \qquad \Rightarrow_\beta$$

$$(\lambda x.\lambda y.xy)(\lambda x.\lambda y.xy) \quad \Rightarrow_\beta$$

$$\lambda y.(\lambda x.\lambda y.xy)y \qquad \Rightarrow_\beta$$

$$\lambda y.\lambda y.yy$$

We have enough evidence by now to understand why substitution, because of the variable capture phenomenon, is considered to be such a tricky and subtle issue by many computer scientists.

We will later see an ingenious technique to transform λ-calculus into combinatory logic which, surprisingly, eliminates the need for substitutions entirely.

Confluence of β-Reduction

Consider the λ-expression $(\lambda f.(\lambda x.fx)y)g$. Note that one has two different ways to apply β-reduction on this λ-expression:

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 the intermediate 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* $\Rightarrow_\beta$):

<u>Theorem.</u> $\Rightarrow_\beta$ *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 $(\lambda x.xx)(\lambda x.xx)$, say omega, known also as the *divergent combinator*. 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.

Exercise 5 *Define λ-calculus formally in Maude. As we noticed, substitution is quite tricky. Instead of assuming that the λ-expressions that are reduced are well-behaved enough so that variable captures do not occur during the β-reduction process, you should define the substitution as a partial operation. In other words, a substitution applies only if it does not lead to a variable capture; you do not need to fix its application by performing appropriate α-conversions. To achieve that, all you need to do is to define the substitution of $(\lambda y.E)[x \leftarrow E']$ when $y \neq x$ as a conditional equation: defined only when $y \notin FV(E')$. Then show that there are λ-expressions that cannot be β-reduced automatically with your definition of λ-calculus, even though they are closed (or combinators) and all the binding variables are initially distinct from each other.*

λ-Calculus as a Programming Language

We have seen how several programming language constructs translate naturally into λ-calculus. Then a natural question arise: can we use λ-calculus as a programming language?

The answer is yes, we can, but we first need to understand how several important programming language features can be systematically captured by λ-calculus, including functions with multiple arguments, booleans, numbers, and recursion.

Currying

Recall from mathematics 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 $X \to Y$. 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, we say that

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 $\lambda yz.(E[x \leftarrow E'])$. Also, since λ-application was defined to be left-associative, $(\lambda xyz.E)E_1 E_2$ β-reduces to $\lambda z.((E[x \leftarrow E_1])[y \leftarrow E_2])$.

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. A syntactic desugaring transformer can apply uncurrying automatically before anything else is defined.

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` $:= \lambda xy.(x\,y\,\texttt{false})$

Exercise 6 *Define the other boolean operations (including at least* `or`*,* `not`*,* `implies`*,* `iff`*, and* `xor`*) as λ-expressions.*

This encoding for booleans is known under the name of *Church booleans*. One can use β-reduction to show, for example, that `and true false` $\Rightarrow_\beta$ `false`. Therefore, `and true false` $\equiv_\beta$ `false`.

One can show relatively easily that the Church booleans have all the desired properties of booleans. Let us, for example, show the associativity of `and`:

$$\text{and (and x y) z} \equiv_\beta \text{x y false z false}$$

$$\text{and x (and y z)} \equiv_\beta \text{x (y z false) false}$$

Obviously, one cannot expect the properties of booleans to hold for any λ-lambda 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.

Exercise 7 *Prove that the Church booleans have all the properties of booleans (the Maude command "*`show module BOOL`*" lists them).*

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

$(\lambda\text{and}.\lambda\text{true}.\lambda\text{false}.\ \text{and true false})$
$\quad\quad((\lambda\text{false}.\lambda xy.\, x\, y\, \text{false})(\lambda xy.y))(\lambda xy.x)(\lambda xy.y).$

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 x and y, it is the case that `1st (pair `x` `y`)` and `2nd (pair `x` `y`)` are β-equivalent

to x and y, respectively.

Fortunately, these can be defined quite easily:

> `pair` := $\lambda xyb.bxy$,
>
> `1st` := $\lambda p.\, p\,$`true`, and
>
> `2nd` := $\lambda p.\, p\,$`false`.

The idea is therefore that `pair` x y gets evaluated to the λ-expression $\lambda b.bxy$, which "freezes" x and y 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:

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

$$1_\lambda \ := \ \lambda sz.sz$$

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

$$3_\lambda \ := \ \lambda sz.s(s(sz)) \qquad \ldots$$

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

```
succ := λnsz.ns(sz)
```

The above says that for a given numeral n, its successor "`succ n`" is the numeral that applies the operation s for n times starting with sz. 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 nsz; therefore, one can define `succ'` $:= λnsz.s(nsz)$.

One may, of course, want to show that `succ` and `succ'` are equal. An interesting observation is that 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, $\lambda sz.ss$. However, they are equal when applied on Church numerals:

Exercise 8 *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.*

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

$$\texttt{plus} := \lambda mnsz.ms(nsz)$$

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:

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

Exercise 9 *Prove that Church numerals form indeed a model of natural numbers, by showing the two properties derived from Peano's axioms above.*

Exercise 10 *Define multiplication on Church numerals and prove its Peano properties.*

Hint. *Multiplication can be defined several different interesting ways.*

Exercise 11 *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.*

Interestingly, 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. More precisely, 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 trick 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:

```
start := pair 0_λ 1_λ,
step  := λp . pair (2nd p) (plus (1st p) (2nd p)),
fibo  := λn . 1st (n step start).
```

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

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

$$\texttt{start} := \texttt{pair } 0_\lambda \ 0_\lambda,$$

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

$$\texttt{pred} := \lambda n \ . \ \texttt{1st } (n \ \texttt{step start}).$$

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? := `λn` . `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 := `λmn` . zero? (sub `m` `n`)`,

`geq := `λmn` . zero? (sub `n` `m`)`,

`equal := `λmn` . and (leq `m` `n`) (geq `m` `n`)`.

Adding Built-ins

As we have discussed, λ-calculus is powerful enough to define many other data-structures and data-types of interest. As it is the case with many other, if not all, *pure* programming paradigms, in order to be usable as a reasonably efficient programming language, λ-calculus needs to provide "built-ins" comprising efficient implementations for frequent data-types and operations on them.

We here only discuss the addition of *built-in integers* to λ-calculus. We say that the new λ-calculus that is obtained this way is *enriched*. Surprisingly, we have very little to do to enrich λ-calculus with builtin integers: we only need to define integers as λ-expressions. In the context of a formal definition of λ-calculus as an equational theory in Maude or any other similar language that already provides efficient equational libraries for integers, one only

needs to transform the already existing definition of λ-calculus, say

```
mod LAMBDA is
  sort Exp .

  ...

endm
```

into a definition of the form

```
mod LAMBDA is
  including INT .
  sort Exp .
  subsort Int < Exp .

  ...

endm
```

importing the builtin module `INT` and then stating that `Int` is a subsort of `Exp`. This way, integers can be used just like any other λ-expressions. One can, of course, write now 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 (so we would "catch" such ill-formed λ-expressions at "runtime".

β-reduction is now itself enriched with the rewriting relation that the builtin integers come with. For example, in `INT`, $7 + 5$ reduces to 12; we write this $7 + 5 \Rightarrow 12$. Then a λ-expression $\lambda x.7 + 5$ reduces immediately to $\lambda x.12$, without applying any β-reduction step but only the reduction that `INT` comes with.

Moreover, β-reduction and `INT`-reduction work together very smoothly. For example, $(\lambda yx.7 + y)5$ first β-reduces to $\lambda x.7 + 5$ and then `INT`-reduces to $\lambda x.12$. In order for this to work, since integers are now constructors for λ-expressions as well, one needs to add one more equation to the definition of substitution:

$$I[x \leftarrow E'] = I, \text{ for any integer } I.$$

Recursion

To understand recursion, one must first understand recursion.

Unknown.

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

```
omega := (λx.xx)(λx.xx),
```

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

But what is a "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 one

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:

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

Note that the "`let rec`" 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. This also explains in some more foundational way why a functional language would report an error when one uses "`let`" instead of "`let rec`".

The foundational question regarding recursion in λ-calculus is therefore the following: how can one 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 translated to the following one: can we find a *fix 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 surprising result:

<u>Fix-Point Theorem.</u> *For any λ-expression F, there is some λ-expression X such that $FX \equiv_\beta X$.*

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

$$
\begin{aligned}
X \quad &= \quad (\lambda x.F(xx))(\lambda x.F(xx)) \\
&\equiv_\beta \quad F((\lambda x.F(xx))(\lambda x.F(xx))) \\
&= \quad FX.
\end{aligned}
$$

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

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

With this, for any λ-expression F, the λ-application YF becomes the fix-point of F; therefore, $F(YF) \equiv_\beta (YF)$. Thus, we have a constructive way to build fix-points for any λ-expression F. Note that F does not even 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 := λf.λx.(if x == 0 then 1 else x * f(x - 1))
```

The recursive definition of factorial is therefore the fix-point of `F`, that is, $Y\mathtt{F}$. It is such a fixed-point λ-expression that the "`let rec`"

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:

```
(YF) 3 ≡β
F (YF) 3 =
(λf.λx.(if x == 0 then 1 else x * f(x - 1))) (YF) 3 ⇒β
if 3 == 0 then 1 else 3 * (YF)(3 - 1) ⇒
3 * ((YF) 2) ≡β
...
6 * ((YF) 0) ≡β
6 * (F (YF) 0) =
6 * ((λf.λx.(if x == 0 then 1 else x * f(x - 1))) (YF) 0) ⇒β
6 * if 0 == 0 then 1 else 0 * (YF)(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 fix-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 SKK; indeed,

$$SKK \equiv_\beta \lambda z.Kz(Kz) = \lambda z.(\lambda xy.x)z(Kz) \equiv_\beta \lambda z.z \equiv_\alpha \lambda x.x.$$

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.

η-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 $\lambda x.Ex = \lambda x.(\lambda y.F)x \equiv_\beta \lambda x.F[y \leftarrow x]$. 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:

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

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 called $\lambda + \eta$.

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 $\{x_1, ..., x_n\}$. 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,

$$\texttt{If } x \in S = x \in S' \texttt{ for any } x \texttt{, then } S = S'.$$

Therefore, in order to show sets S and S' equal, one can first "extend" them (regarded as syntactic terms) by applying them the membership operator. In most cases the new task is easier 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 $E \equiv_{\beta\text{ext}} E'$ 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 $\lambda + \text{ext}$.

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

Theorem. $\lambda + \eta$ _is equivalent to_ $\lambda + \text{ext}$.

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 $\lambda + \texttt{ext}$, 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 $\texttt{ext}$. 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, $\lambda + \texttt{ext}$ 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 $\texttt{ext}$ 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. $\qquad\qquad\square$

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. $\lambda + $ `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: $\lambda + \texttt{ext}$ implies $\texttt{CL} + \texttt{ext}$. We first need to show how the syntax of $\texttt{CL} + \texttt{ext}$ translates into that of $\lambda + \texttt{ext}$. 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 $\texttt{CL} + \texttt{ext}$ hold under this translation: they can be immediately proved by β-equivalence. We also need to show that the extensionality in $\texttt{CL} + \texttt{ext}$ holds under the above translation: this is obvious, because it is exactly the same as the extensionality in $\lambda + \texttt{ext}$.

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 $\lambda_{\_}._{\_} : \textit{Var} \rightarrow \textit{Exp}$. 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 12 *(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:

$(\lambda x.\rho\rho')z \equiv_\beta (\rho[x \leftarrow z])(\rho'[x \leftarrow z]) \equiv_\beta ((\lambda x.\rho)z)((\lambda x.\rho')z)) \equiv_\beta (\lambda xyz.xz(yz))(\lambda x.\rho)(\lambda x.\rho')z = S(\lambda x.\rho)(\lambda x.\rho')z$, so by extensionality, $\lambda x.\rho\rho' \equiv_{\beta \mathrm{ext}} S(\lambda x.\rho)(\lambda x.\rho')$.

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 can do with λ-claculus and `ext` can also do with with its "sub-calculus" `CL` + `ext`. This is not hard, but rather technical.

Exercise 13 *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 5, but whose translation in* CL *can be reduced with the two equations in* CL.