computable by a Turing machine. Independently, Alonzo Church answered negatively assuming that "calculable" meant a function expressible in the λ-calculus.

6.1.1 λ-Notation: Main Ideas

The λ-calculus is built around the idea of expressing a calculus of functions, where it is not necessary to assign names to functions, i.e., where functions can be expressed anonymously. Conceptually, this amounts to have the possibility of:

- forming (anonymous) functions by *abstraction* over names in an expression; and
- *applying* a function to an argument

Building on the two basic considerations above, Church developed a theory of functions based on rules for computation, as opposed to the classical set-theoretic view of functions as sets of pairs (argument, result).

Example 6.1. Let us start with a simple example from arithmetic. Take a polynomial such as

$$x^2 - 2x + 5.$$

What is the value of the above expression when x is replaced by 2? We compute the result by plugging in '2' for 'x' in the expression to get

$$2^2 - 2 \times 2 + 5 = 5.$$

In λ-notation, when we want to express that the value of an expression depends on some value to be plugged in, we use abstraction. Syntactically, this corresponds to prefix the expression by the special symbol λ and the name of the formal parameter, as, e.g., in:

$$\lambda x.\ (x^2 - 2x + 5)$$

The informal reading is:

> wait for a value v to replace x and then compute $v^2 - 2v + 5$.

We want to be able to pass some actual parameter to the function above, i.e., to apply the function to some value v. To this aim, we denote application by juxtaposition:

$$(\lambda x.\ (x^2 - 2x + 5))\ 2$$

means that the function $(\lambda x.\ (x^2 - 2x + 5))$ is applied to 2 (i.e., that the actual parameter 2 must replace the occurrences of the formal parameter x in $x^2 - 2x + 5$, to obtain $2^2 - 2 \times 2 + 5 = 5$.)

Note that:

- by writing $\lambda x.\ t$ we are declaring x as a formal parameter appearing in t;
- the symbol λ has no particular meaning (any other symbol could have been used);

- we say that λx 'binds' the (occurrences of the) variable x in t;
- the scope of the formal parameter x is just t; if x occurs also "outside" t, then it refers to another (homonymous) identifier.

Example 6.2. Let us consider another example:

$$(\lambda x.\ \lambda y.\ (x^2 - 2y + 5))\ 2$$

This time we have a function that is waiting for two arguments (first x, then y), but to which we pass one value (2). We have

$$(\lambda x.\ \lambda y.\ (x^2 - 2y + 5))\ 2 = \lambda y.\ (2^2 - 2y + 5) = \lambda y.\ (9 - 2y)$$

that is, the result of applying $\lambda x.\ \lambda y.\ (x^2 - 2y + 5)$ to 2 is still a function $(\lambda y.\ (9 - 2y))$.

In λ-calculus we can pass functions as arguments and return functions as results.

Example 6.3. Take the term $\lambda f.\ (f\ 2)$: it waits for a function f that will be applied to the value 2. If we pass the function $(\lambda x.\ \lambda y.\ (x^2 - 2y + 5))$ to $\lambda f.\ (f\ 2)$, written:

$$(\lambda f.\ (f\ 2))\ (\lambda x.\ \lambda y.\ (x^2 - 2y + 5))$$

then we get the function $\lambda y.\ (9 - 2y)$ as a result.

Definition 6.1 (Lambda terms). We define *lambda terms* as the terms generated by the grammar:

$$t \quad ::= \quad x \quad | \quad \lambda x.\, t \quad | \quad (t_0\ t_1) \quad | \quad t \to (t_0, t_1)$$

Where x is a variable.

As we can see the lambda notation is very simple, it has four constructs:

- x: is a simple *variable*.
- $\lambda x.\ t$: is the *lambda abstraction* which allows to define anonymous functions.
- $t_0\ t_1$: is the *application* of a function t_0 to its argument t_1.
- $t \to t_0, t_1$ is the conditional operator, i.e. the "if-then-else" construct in lambda notation.

Note that we omit some parentheses when no ambiguity can arise.

Lambda abstraction $\lambda x.\ t$ is the main feature. It allows to define functions, where x represents the parameter of the function and t is the lambda term which represents the body of the function. For example the term $\lambda x.\ x$ is the identity function.

Note that while we can have different terms t and t' that define the same function, Church proved that the problem of deciding whether $t = t'$ is undecidable.

Definition 6.2 (Conditional expressions). Let t, t_0 and t_1 be three lambda terms, we define:

$$t \to t_0, t_1 = \begin{cases} t_0 & \text{if } t = true \\ t_1 & \text{if } t = false \end{cases}$$

All the notions used in this definition, like "true" and "false" can be formalised in lambda notation only, by using lambda abstraction, as shown in Section 6.1.1.1 for the interested reader. In the following we will take the liberty to assume that data types such as integers and booleans are available in the lambda-notation as well as the usual operations on them.

Remark 6.1 (Associativity of abstraction and application). In the following, to limit the number of parentheses and keep the notation more readable, we assume that application is left-associative, and lambda-abstraction is right-associative, i.e.,

$$\begin{array}{ll} t_1\ t_2\ t_3\ t_4 & \text{is read as } (((t_1\ t_2)\ t_3)\ t_4) \\ \lambda x_1.\ \lambda x_2.\ \lambda x_3.\ \lambda x_4.\ t & \text{is read as } \lambda x_1.\ (\lambda x_2.\ (\lambda x_3.\ (\lambda x_4.\ t))) \end{array}$$

Remark 6.2 (Precedence of application). We will also assume that application has precedence over abstraction, i.e.:

$$\lambda x.\ t\ t' = \lambda x.\ (t\ t')$$

6.1.1.1 λ-Notation: Booleans and Church Numerals

In the above examples, we have enriched standard arithmetic expressions with abstraction and application. In general, it would be possible to encode booleans and numbers (and operations over them) just using abstraction and application.

For example, let us consider the following terms:

$$T \stackrel{\text{def}}{=} \lambda x.\ \lambda y.\ x$$
$$F \stackrel{\text{def}}{=} \lambda x.\ \lambda y.\ y$$

We can assume that T represents true and F represents false.

Under this convention, we can define the usual logical operations by letting:

$$\text{AND} \stackrel{\text{def}}{=} \lambda p.\ \lambda q.\ p\ q\ p$$
$$\text{OR} \stackrel{\text{def}}{=} \lambda p.\ \lambda q.\ p\ p\ q$$
$$\text{NOT} \stackrel{\text{def}}{=} \lambda p.\ \lambda x.\ \lambda y.\ p\ y\ x$$

Now suppose that P will reduce either to T or to F. The expression $P\ A\ B$ can be read as 'if P then A else B'.

For natural numbers, we can adopt the convention that the number n is represented by a function that takes a function f and an argument x and applies f to x for n times consecutively. For example:

$$
\begin{aligned}
0 &\stackrel{\text{def}}{=} \lambda f.\ \lambda x.\ x \\
1 &\stackrel{\text{def}}{=} \lambda f.\ \lambda x.\ f\ x \\
2 &\stackrel{\text{def}}{=} \lambda f.\ \lambda x.\ f\ (f\ x) \\
&\dots
\end{aligned}
$$

Then, the operations for successor, sum, multiplication can be defined by letting:

$$
\begin{aligned}
\text{SUCC} &\stackrel{\text{def}}{=} \lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x) \\
\text{SUM} &\stackrel{\text{def}}{=} \lambda n.\ \lambda m.\ \lambda f.\ \lambda x.\ m\ f\ (n\ f\ x) \\
\text{MUL} &\stackrel{\text{def}}{=} \lambda n.\ \lambda m.\ \lambda f.\ n\ (m\ f)
\end{aligned}
$$

6.1.2 Alpha-Conversion, Beta-Rule and Capture-Avoiding Substitution

The names of the formal parameters we choose for a given function should not matter. Therefore, any two expressions that differ just for the particular choice of λ-abstracted variables and have the same structure otherwise, should be considered as equal.

For example, we do not want to distinguish between the terms

$$\lambda x.\ (x^2 - 2x + 5) \qquad \lambda y.\ (y^2 - 2y + 5)$$

On the other hand, the expressions

$$x^2 - 2x + 5 \qquad y^2 - 2y + 5$$

must be distinguished, because depending on the context where they are used, the symbols x and y could have a different meaning.

We say that two terms are α-convertible if one is obtained from the other by renaming some λ-abstracted variables. We call *free* the variables x whose occurrences are not under the scope of a λ binder.

Definition 6.3 (Free variables). The set of free variables occurring in a term is defined by structural recursion:

$$
\begin{aligned}
\text{fv}(x) &\stackrel{\text{def}}{=} \{x\} \\
\text{fv}(\lambda x.\ t) &\stackrel{\text{def}}{=} \text{fv}(t) \setminus \{x\} \\
\text{fv}(t_0\ t_1) &\stackrel{\text{def}}{=} \text{fv}(t_0) \cup \text{fv}(t_1) \\
\text{fv}(t \to t_0, t_1) &\stackrel{\text{def}}{=} \text{fv}(t) \cup \text{fv}(t_0) \cup \text{fv}(t_1)
\end{aligned}
$$

The second equation highlights that the lambda abstraction is a binding operator.

Definition 6.4 (Alpha-conversion). We define *α-conversion* as the equivalence induced by letting

$$\lambda x.\, t = \lambda y.\, (t[^{y}/_{x}]) \quad \text{if } y \notin \mathrm{fv}(t)$$

where $t[^{y}/_{x}]$ denotes the substitution of x with y applied to the term t.

Note the side condition $y \notin \mathrm{fv}(t)$, which is needed to avoid 'capturing' other free variables appearing in t.

For example:

$$\lambda z.\, z^2 - 2y + 5 = \lambda x.\, x^2 - 2y + 5 \neq \lambda y.\, y^2 - 2y + 5$$

We have now all ingredients to define the basic computational rule, called β-rule, which explains how to apply a function to an argument:

Definition 6.5 (Beta-rule). Let t, t' be two lambda terms we define:

$$(\lambda x.\, t')\, t = t'[^{t}/_{x}]$$

this axiom is called β-rule.

In defining alpha-conversion and the beta-rule we have used substitutions like $[^{y}/_{x}]$ and $[^{t}/_{x}]$. Let us now try to formalise the notion of substitution by structural recursion. What is wrong with the following naive attempt?

$$y[^{t}/_{x}] \stackrel{\text{def}}{=} \begin{cases} t & \text{if } y = x \\ y & \text{if } y \neq x \end{cases}$$

$$(\lambda y.\, t')[^{t}/_{x}] \stackrel{\text{def}}{=} \begin{cases} \lambda y.\, t' & \text{if } y = x \\ \lambda y.\, (t'[^{t}/_{x}]) & \text{if } y \neq x \end{cases}$$

$$(t_0\ t_1)[^{t}/_{x}] \stackrel{\text{def}}{=} (t_0[^{t}/_{x}])\ (t_1[^{t}/_{x}])$$

$$(t' \to t_0, t_1)[^{t}/_{x}] \stackrel{\text{def}}{=} (t'[^{t}/_{x}]) \to (t_0[^{t}/_{x}]), (t_1[^{t}/_{x}])$$

Example 6.4 (Substitution, without alpha-renaming). Consider the terms

$$t \stackrel{\text{def}}{=} \lambda x.\, \lambda y.\, (x^2 - 2y + 5) \qquad t' \stackrel{\text{def}}{=} y.$$

and apply t to t':

$$\begin{aligned} t\ t' &= (\lambda x.\, \lambda y.\, (x^2 - 2y + 5))\ y \\ &= (\lambda y.\, (x^2 - 2y + 5))[^{y}/_{x}] \\ &= \lambda y.\, ((x^2 - 2y + 5)[^{y}/_{x}]) \\ &= \lambda y.\, (y^2 - 2y + 5) \end{aligned}$$

It happens that the free variable $y \in \mathrm{fv}(t\ t')$ has been 'captured' by the lambda-abstraction λy. Instead, free variables occurring in t should remain free during the application of the substitution $[^{t}/_{x}]$.

Thus we need to correct the above version of substitution for the case related to $(\lambda y.\ t')[{}^{t}/_{x}]$ by applying first the alpha-conversion to $\lambda y.\ t'$ (to make sure that if $y \in \mathrm{fv}(t)$, then the free occurrences of y in t will not be captured by λy when replacing x in t') and then the substitution $[{}^{t}/_{x}]$. Formally, we let:

Definition 6.6 (Capture-avoiding substitution). Let t, t', t_0 and t_1 be four lambda terms, we define:

$$
\begin{aligned}
y[{}^{t}/_{x}] &\stackrel{\text{def}}{=} \begin{cases} t & \text{if } y = x \\ y & \text{if } y \neq x \end{cases} \\
(\lambda y.\ t')[{}^{t}/_{x}] &\stackrel{\text{def}}{=} \lambda z.\ ((t'[{}^{z}/_{y}])[{}^{t}/_{x}]) \qquad \text{if } z \notin \mathrm{fv}(\lambda y.\ t') \cup \mathrm{fv}(t) \cup \{x\} \\
(t_0\ t_1)[{}^{t}/_{x}] &\stackrel{\text{def}}{=} (t_0[{}^{t}/_{x}])\ (t_1[{}^{t}/_{x}]) \\
(t' \to t_0, t_1)[{}^{t}/_{x}] &\stackrel{\text{def}}{=} (t'[{}^{t}/_{x}]) \to (t_0[{}^{t}/_{x}]), (t_1[{}^{t}/_{x}])
\end{aligned}
$$

Note that the matter of names is not so trivial. In the second equation we first rename y in t' with a fresh name z, then proceed with the substitution of x with t. As explained, this solution is motivated by the fact that y might not be free in t, but it introduces some non-determinism in the equations due to the arbitrary nature of the new name z. This non-determinism immediately disappear if we regard the terms up to the alpha-conversion equivalence, as previously introduced. Obviously α-conversion and substitution should be defined at the same time to avoid circularity. By using the α-conversion we can prove statements like $\lambda x.\ x = \lambda y.\ y$.

Example 6.5 (Application with alpha-renaming). Consider the terms t, t' from Example 6.4:

$$
\begin{aligned}
t\ t' &= (\lambda x.\ \lambda y.\ (x^2 - 2y + 5))\ y \\
&= (\lambda y.\ (x^2 - 2y + 5))[{}^{y}/_{x}] \\
&= \lambda z.\ ((x^2 - 2y + 5)[{}^{z}/_{y}][{}^{y}/_{x}]) \\
&= \lambda z.\ ((x^2 - 2z + 5)[{}^{y}/_{x}]) \\
&= \lambda z.\ (y^2 - 2z + 5)
\end{aligned}
$$

Finally we introduce some notational conventions for omitting parentheses when defining the domains and codomains of functions:

$$
\begin{aligned}
A \to B \times C &= A \to (B \times C) & \qquad A \times B \times C &= (A \times B) \times C \\
A \times B \to C &= (A \times B) \to C & \qquad A \to B \to C &= A \to (B \to C)
\end{aligned}
$$

6.2 Denotational Semantics of IMP

As we said we will use lambda notation as meta-language; this means that we will express the semantics of IMP by translating IMP syntax to lambda terms.