21 The Untyped λ-Calculus

In this chapter, we study the premier example of a uni-typed programming language, the *(untyped)* λ*-calculus*. This formalism was introduced by Church in the 1930s as a universal language of computable functions. It is distinctive for its austere elegance. The λ-calculus has but one "feature," the higher-order function. Everything is a function, hence every expression may be applied to an argument, which must itself be a function, with the result also being a function. To borrow a turn of phrase, in the λ-calculus it's functions all the way down.

21.1 The λ-Calculus

The abstract syntax of the untyped λ-calculus, called $\mathbf{\Lambda}$, is given by the following grammar:

Exp	u	$::=$	x	x	variable
			$\lambda(x.u)$	$\lambda\,(x)\,u$	λ-abstraction
			$\mathsf{ap}(u_1;u_2)$	$u_1(u_2)$	application

The statics of $\mathbf{\Lambda}$ is defined by general hypothetical judgments of the form $x_1\ \mathsf{ok}, \ldots, x_n\ \mathsf{ok} \vdash u\ \mathsf{ok}$, stating that u is a well-formed expression involving the variables $x_1, \ldots, x_n$. (As usual, we omit explicit mention of the variables when they can be determined from the form of the hypotheses.) This relation is inductively defined by the following rules:

$$\frac{}{\Gamma, x\ \mathsf{ok} \vdash x\ \mathsf{ok}} \tag{21.1a}$$

$$\frac{\Gamma \vdash u_1\ \mathsf{ok} \quad \Gamma \vdash u_2\ \mathsf{ok}}{\Gamma \vdash u_1(u_2)\ \mathsf{ok}} \tag{21.1b}$$

$$\frac{\Gamma, x\ \mathsf{ok} \vdash u\ \mathsf{ok}}{\Gamma \vdash \lambda\,(x)\,u\ \mathsf{ok}} \tag{21.1c}$$

The dynamics of $\mathbf{\Lambda}$ is given equationally, rather than via a transition system. Definitional equality for $\mathbf{\Lambda}$ is a judgment of the form $\Gamma \vdash u \equiv u'$, where $\Gamma = x_1\ \mathsf{ok}, \ldots, x_n\ \mathsf{ok}$ for some $n \geq 0$, and u and u' are terms having at most the variables $x_1, \ldots, x_n$ free. It is inductively defined by the following rules:

$$\frac{}{\Gamma, u\ \mathsf{ok} \vdash u \equiv u} \tag{21.2a}$$

https://doi.org/10.1017/CBO9781316576892.023 Published online by Cambridge University Press

$$\frac{\Gamma \vdash u \equiv u'}{\Gamma \vdash u' \equiv u} \tag{21.2b}$$

$$\frac{\Gamma \vdash u \equiv u' \quad \Gamma \vdash u' \equiv u''}{\Gamma \vdash u \equiv u''} \tag{21.2c}$$

$$\frac{\Gamma \vdash u_1 \equiv u_1' \quad \Gamma \vdash u_2 \equiv u_2'}{\Gamma \vdash u_1(u_2) \equiv u_1'(u_2')} \tag{21.2d}$$

$$\frac{\Gamma, x\ \mathsf{ok} \vdash u \equiv u'}{\Gamma \vdash \lambda\,(x)\,u \equiv \lambda\,(x)\,u'} \tag{21.2e}$$

$$\frac{\Gamma, x\ \mathsf{ok} \vdash u_2\ \mathsf{ok} \quad \Gamma \vdash u_1\ \mathsf{ok}}{\Gamma \vdash (\lambda\,(x)\,u_2)(u_1) \equiv [u_1/x]u_2} \tag{21.2f}$$

We often write just $u \equiv u'$ when the variables involved need not be emphasized or are clear from context.

21.2 Definability

Interest in the untyped λ-calculus stems from its surprising expressiveness. It is a *Turing-complete* language in the sense that it has the same capability to express computations on the natural numbers as does any other known programming language. Church's Law states that any conceivable notion of computable function on the natural numbers is equivalent to the λ-calculus. This assertion is true for all *known* means of defining computable functions on the natural numbers. The force of Church's Law is that it postulates that all future notions of computation will be equivalent in expressive power (measured by definability of functions on the natural numbers) to the λ-calculus. Church's Law is therefore a *scientific law* in the same sense as, say, Newton's Law of Universal Gravitation, which predicts the outcome of all future measurements of the acceleration in a gravitational field.[1]

We will sketch a proof that the untyped λ-calculus is as powerful as the language PCF described in Chapter 19. The main idea is to show that the PCF primitives for manipulating the natural numbers are definable in the untyped λ-calculus. In particular, we must show that the natural numbers are definable as λ-terms in such a way that case analysis, which discriminates between zero and non-zero numbers, is definable. The principal difficulty is with computing the predecessor of a number, which requires a bit of cleverness. Finally, we show how to represent general recursion, completing the proof.

The first task is to represent the natural numbers as certain λ-terms, called the *Church numerals*.

$$\overline{0} \triangleq \lambda\,(b)\,\lambda\,(s)\,b \tag{21.3a}$$

$$\overline{n+1} \triangleq \lambda\,(b)\,\lambda\,(s)\,s(\overline{n}(b)(s)) \tag{21.3b}$$

It follows that

$$\overline{n}(u_1)(u_2) \equiv u_2(\ldots(u_2(u_1))),$$

https://doi.org/10.1017/CBO9781316576892.023 Published online by Cambridge University Press

the n-fold application of u_2 to u_1. That is, $\overline{n}$ iterates its second argument (the induction step) n times, starting with its first argument (the basis).

Using this definition, it is not difficult to define the basic functions of arithmetic. For example, successor, addition, and multiplication are defined by the following untyped λ-terms:

$$\mathtt{succ} \triangleq \lambda\,(x)\,\lambda\,(b)\,\lambda\,(s)\,s(x(b)(s)) \tag{21.4}$$

$$\mathtt{plus} \triangleq \lambda\,(x)\,\lambda\,(y)\,y(x)(\mathtt{succ}) \tag{21.5}$$

$$\mathtt{times} \triangleq \lambda\,(x)\,\lambda\,(y)\,y(\overline{0})(\mathtt{plus}(x)) \tag{21.6}$$

It is easy to check that $\mathtt{succ}(\overline{n}) \equiv \overline{n+1}$, and that similar correctness conditions hold for the representations of addition and multiplication.

To define $\mathtt{ifz}\{u_0; x.u_1\}(u)$ requires a bit of ingenuity. The key is to define the "cut-off predecessor," `pred`, such that

$$\mathtt{pred}(\overline{0}) \equiv \overline{0} \tag{21.7}$$

$$\mathtt{pred}(\overline{n+1}) \equiv \overline{n}. \tag{21.8}$$

To compute the predecessor using Church numerals, we must show how to compute the result for $\overline{n+1}$ in terms of its value for $\overline{n}$. At first glance, this seems simple—just take the successor—until we consider the base case, in which we define the predecessor of $\overline{0}$ to be $\overline{0}$. This formulation invalidates the obvious strategy of taking successors at inductive steps, and necessitates some other approach.

What to do? A useful intuition is to think of the computation in terms of a pair of "shift registers" satisfying the invariant that on the nth iteration the registers contain the predecessor of n and n itself, respectively. Given the result for n, namely the pair $(n-1, n)$, we pass to the result for $n+1$ by shifting left and incrementing to obtain $(n, n+1)$. For the base case, we initialize the registers with $(0, 0)$, reflecting the stipulation that the predecessor of zero be zero. To compute the predecessor of n, we compute the pair $(n-1, n)$ by this method, and return the first component.

To make this precise, we must first define a Church-style representation of ordered pairs.

$$\langle u_1, u_2\rangle \triangleq \lambda\,(f)\,f(u_1)(u_2) \tag{21.9}$$

$$u \cdot \mathtt{l} \triangleq u(\lambda\,(x)\,\lambda\,(y)\,x) \tag{21.10}$$

$$u \cdot \mathtt{r} \triangleq u(\lambda\,(x)\,\lambda\,(y)\,y) \tag{21.11}$$

It is easy to check that under this encoding $\langle u_1, u_2\rangle \cdot \mathtt{l} \equiv u_1$, and that a similar equivalence holds for the second projection. We may now define the required representation, u_p, of the predecessor function:

$$u'_p \triangleq \lambda\,(x)\,x(\langle\overline{0}, \overline{0}\rangle)(\lambda\,(y)\,\langle y \cdot \mathtt{r}, \mathtt{succ}\,(y \cdot \mathtt{r})\rangle) \tag{21.12}$$

$$u_p \triangleq \lambda\,(x)\,u'_p(x) \cdot \mathtt{l} \tag{21.13}$$

It is easy to check that this gives us the required behavior. Finally, define $\mathtt{ifz}\{u_0; x.u_1\}(u)$ to be the untyped term

$$u(u_0)(\lambda\,(\_)\,[u_p(u)/x]u_1).$$

https://doi.org/10.1017/CBO9781316576892.023 Published online by Cambridge University Press

This definition gives us all the apparatus of PCF, apart from general recursion. But general recursion is also definable in $\mathbf{\Lambda}$ using a *fixed point combinator*. There are many choices of fixed point combinator, of which the best known is the $\mathtt{Y}$ *combinator*:

$$\mathtt{Y} \triangleq \lambda\,(F)\,(\lambda\,(f)\,F(f(f)))(\lambda\,(f)\,F(f(f))).$$

It is easy to check that

$$\mathtt{Y}(F) \equiv F(\mathtt{Y}(F)).$$

Using the $\mathtt{Y}$ combinator, we may define general recursion by writing $\mathtt{Y}(\lambda\,(x)\,u)$, where x stands for the recursive expression itself.

Although it is clear that $\mathtt{Y}$ as just defined computes a fixed point of its argument, it is probably less clear why it works or how we might have invented it in the first place. The main idea is quite simple. If a function is recursive, it is given an extra first argument, which is arranged at call sites to be the function itself. Whenever we wish to call a self-referential function with an argument, we apply the function first to itself and then to its argument; this protocol is imposed on both the "external" calls to the function and on the "internal" calls that the function may make to itself. For this reason, the first argument is often called `this` or `self`, to remind you that it will be, by convention, bound to the function itself.

With this in mind, it is easy to see how to derive the definition of $\mathtt{Y}$. If F is the function whose fixed point we seek, then the function $F' = \lambda\,(f)\,F(f(f))$ is a variant of F in which the self-application convention has been imposed internally by substituting for each occurrence of f in $F(f)$ the self-application $f(f)$. Now check that $F'(F') \equiv F(F'(F'))$, so that $F'(F')$ is the desired fixed point of F. Expanding the definition of F', we have derived that the desired fixed point of F is

$$\lambda\,(f)\,F(f(f))(\lambda\,(f)\,F(f(f))).$$

To finish the derivation, we need only note that nothing depends on the particular choice of F, which means that we can compute a fixed point for F uniformly in F. That is, we may define a *single* function, the term $\mathtt{Y}$ as defined above, that computes the fixed point of *any* F.

21.3 Scott's Theorem

Scott's Theorem states that definitional equality for the untyped λ-calculus is undecidable: there is no algorithm to determine whether two untyped terms are definitionally equal. The proof uses the concept of *inseparability*. Any two properties, $\mathcal{A}_0$ and $\mathcal{A}_1$, of λ-terms are *inseparable* if there is no decidable property, $\mathcal{B}$, such that $\mathcal{A}_0\ u$ implies that $\mathcal{B}\ u$ holds, and $\mathcal{A}_1\ u$ implies that $\mathcal{B}\ u$ does *not* hold. We say that a property, $\mathcal{A}$, of untyped terms is *behavioral* iff whenever $u \equiv u'$, then $\mathcal{A}\ u$ iff $\mathcal{A}\ u'$.

https://doi.org/10.1017/CBO9781316576892.023 Published online by Cambridge University Press

The proof of Scott's Theorem decomposes into two parts:

1. For any untyped λ-term u, we may find an untyped term v such that $u(\overline{\ulcorner v \urcorner}) \equiv v$, where $\ulcorner v \urcorner$ is the Gödel number of v, and $\overline{\ulcorner v \urcorner}$ is its representation as a Church numeral. (See Chapter 9 for a discussion of Gödel-numbering.)
2. Any two non-trivial[2] behavioral properties $\mathcal{A}_0$ and $\mathcal{A}_1$ of untyped terms are inseparable.

Lemma 21.1. *For any u, there exists v such that $u(\overline{\ulcorner v \urcorner}) \equiv v$.*

Proof Sketch The proof relies on the definability of the following two operations in the untyped λ-calculus:

1. $\mathtt{ap}(\overline{\ulcorner u_1 \urcorner})(\overline{\ulcorner u_2 \urcorner}) \equiv \overline{\ulcorner u_1(u_2) \urcorner}$.
2. $\mathtt{nm}(\overline{n}) \equiv \overline{\ulcorner \overline{n} \urcorner}$.

Intuitively, the first takes the representations of two untyped terms and builds the representation of the application of one to the other. The second takes a numeral for n, and yields the representation of the Church numeral $\overline{n}$. Given these, we may find the required term v by defining $v \triangleq w(\overline{\ulcorner w \urcorner})$, where $w \triangleq \lambda\,(x)\,u(\mathtt{ap}(x)(\mathtt{nm}(x)))$. We have

$$\begin{aligned} v &= w(\overline{\ulcorner w \urcorner}) \\ &\equiv u(\mathtt{ap}(\overline{\ulcorner w \urcorner})(\mathtt{nm}(\overline{\ulcorner w \urcorner}))) \\ &\equiv u(\overline{\ulcorner w(\overline{\ulcorner w \urcorner}) \urcorner}) \\ &\equiv u(\overline{\ulcorner v \urcorner}). \end{aligned}$$

The definition is very similar to that of $\mathtt{Y}(u)$, except that u takes as input the representation of a term, and we find a v such that, when applied to the representation of v, the term u yields v itself. □

Lemma 21.2. *Suppose that $\mathcal{A}_0$ and $\mathcal{A}_1$ are two non-trivial behavioral properties of untyped terms. Then there is no untyped term w such that*

1. *For every u, either $w(\overline{\ulcorner u \urcorner}) \equiv \overline{0}$ or $w(\overline{\ulcorner u \urcorner}) \equiv \overline{1}$.*
2. *If $\mathcal{A}_0\ u$, then $w(\overline{\ulcorner u \urcorner}) \equiv \overline{0}$.*
3. *If $\mathcal{A}_1\ u$, then $w(\overline{\ulcorner u \urcorner}) \equiv \overline{1}$.*

Proof Suppose there is such an untyped term w. Let v be the untyped term

$$\lambda\,(x)\,\mathtt{ifz}\{u_1; \_.u_0\}(w(x)),$$

where u_0 and u_1 are chosen such that $\mathcal{A}_0\ u_0$ and $\mathcal{A}_1\ u_1$. (Such a choice must exist by non-triviality of the properties.) By Lemma 21.1 there is an untyped term t such that $v(\overline{\ulcorner t \urcorner}) \equiv t$. If $w(\overline{\ulcorner t \urcorner}) \equiv \overline{0}$, then $t \equiv v(\overline{\ulcorner t \urcorner}) \equiv u_1$, and so $\mathcal{A}_1\ t$, because $\mathcal{A}_1$ is behavioral and $\mathcal{A}_1\ u_1$.

https://doi.org/10.1017/CBO9781316576892.023 Published online by Cambridge University Press

But then $w(\overline{\ulcorner t \urcorner}) \equiv \overline{1}$ by the defining properties of w, which is a contradiction. Similarly, if $w(\overline{\ulcorner t \urcorner}) \equiv \overline{1}$, then $\mathcal{A}_0\ t$, and hence $w(\overline{\ulcorner t \urcorner}) \equiv \overline{0}$, again a contradiction. □

Corollary 21.3. *There is no algorithm to decide whether $u \equiv u'$.*

Proof For fixed u, the property $\mathcal{E}_u\ u'$ defined by $u' \equiv u$ is a non-trivial behavioral property of untyped terms. So it is inseparable from its negation, and hence is undecidable. □

21.4 Untyped Means Uni-Typed

The untyped λ-calculus can be faithfully embedded in a typed language with recursive types. Thus, every untyped λ-term has a representation as a typed expression in such a way that execution of the representation of a λ-term corresponds to execution of the term itself. This embedding is *not* a matter of writing an interpreter for the λ-calculus in **FPC**, but rather a direct representation of untyped λ-terms as typed expressions in a language with recursive types.

The key observation is that the *untyped* λ-calculus is really the *uni-typed* λ-calculus. It is not the *absence* of types that gives it its power, but rather that it has *only one* type, the recursive type

$$D \triangleq \mathtt{rec}\, t\, \mathtt{is}\, t \rightharpoonup t.$$

A value of type D is of the form $\mathtt{fold}(e)$ where e is a value of type $D \rightharpoonup D$—a function whose domain and range are both D. Any such function can be regarded as a value of type D by "folding", and any value of type D can be turned into a function by "unfolding". As usual, a recursive type is a solution to a type equation, which in the present case is the equation

$$D \cong D \rightharpoonup D.$$

This isomorphism specifies that D is a type that is isomorphic to the space of partial functions on D itself, which is impossible if types are just sets.

This isomorphism leads to the following translation, of **Λ** into **FPC**:

$$x^\dagger \triangleq x \tag{21.14a}$$

$$\lambda\,(x)\,u^\dagger \triangleq \mathtt{fold}(\lambda\,(x : D)\,u^\dagger) \tag{21.14b}$$

$$u_1(u_2)^\dagger \triangleq \mathtt{unfold}(u_1^\dagger)(u_2^\dagger) \tag{21.14c}$$

Note that the embedding of a λ-abstraction is a value, and that the embedding of an application exposes the function being applied by unfolding the recursive type. And so

https://doi.org/10.1017/CBO9781316576892.023 Published online by Cambridge University Press

we have

$$\begin{aligned}\lambda\,(x)\,u_1(u_2)^\dagger &= \mathtt{unfold}(\mathtt{fold}(\lambda\,(x:D)\,u_1^\dagger))(u_2^\dagger)\\ &\equiv \lambda\,(x:D)\,u_1^\dagger(u_2^\dagger)\\ &\equiv [u_2^\dagger/x]u_1^\dagger\\ &= ([u_2/x]u_1)^\dagger.\end{aligned}$$

The last step, stating that the embedding commutes with substitution, is proved by induction on the structure of u_1. Thus β-reduction is implemented by evaluation of the embedded terms.

Thus, we see that the canonical *untyped* language, $\mathbf{\Lambda}$, which by dint of terminology stands in opposition to *typed* languages, turns out to be but a typed language after all. Rather than eliminating types, an untyped language consolidates an infinite collection of types into a single recursive type. Doing so renders static type checking trivial, at the cost of incurring dynamic overhead to coerce values to and from the recursive type. In Chapter 22, we will take this a step further by admitting many different types of data values (not just functions), each of which is a component of a "master" recursive type. This generalization shows that so-called *dynamically typed* languages are, in fact, *statically typed*. Thus, this traditional distinction cannot be considered an opposition, because dynamic languages are but particular forms of static languages in which undue emphasis is placed on a single recursive type.

21.5 Notes

The untyped λ-calculus was introduced by Church (1941) as a formalization of the informal concept of a computable function. Unlike the well-known machine models, such as the Turing machine or the random access machine, the λ-calculus codifies mathematical and programming practice. Barendregt (1984) is the definitive reference for all aspects of the untyped λ-calculus; the proof of Scott's theorem is adapted from Barendregt's account. Scott (1980a) gave the first model of the untyped λ-calculus in terms of an elegant theory of recursive types. This construction underlies Scott's apt description of the λ-calculus as "uni-typed," rather than "untyped." The idea to characterize Church's Law as such was communicated to the author, independently of each other, by Robert L. Constable and Mark Lillibridge.

Exercises

21.1. Define an encoding of finite products as defined in Chapter 10 in $\mathbf{\Lambda}$.

21.2. Define the factorial function in $\mathbf{\Lambda}$ two ways, one without using $\mathtt{Y}$, and one using $\mathtt{Y}$. In both cases, show that your solution, u, has the property that $u(\overline{n}) \equiv \overline{n!}$.

https://doi.org/10.1017/CBO9781316576892.023 Published online by Cambridge University Press

21.3. Define the "Church booleans" in $\mathbf{\Lambda}$ by defining terms `true` and `false` such that
(a) $\texttt{true}(u_1)(u_2) \equiv u_1$.
(b) $\texttt{false}(u_1)(u_2) \equiv u_2$.
What is the encoding of $\texttt{if}\, u\, \texttt{then}\, u_1\, \texttt{else}\, u_2$?

21.4. Define an encoding of finite sums as defined in Chapter 11 in $\mathbf{\Lambda}$.

21.5. Define an encoding of finite lists of natural numbers as defined in Chapter 15 in $\mathbf{\Lambda}$.

21.6. Define an encoding of the infinite streams of natural numbers as defined in Chapter 15 in $\mathbf{\Lambda}$.

21.7. Show that $\mathbf{\Lambda}$ can be "compiled" to `sk`-combinators using bracket abstraction (see Exercises **3.4** and **3.5**. Define a translation u^* from $\mathbf{\Lambda}$ into `sk` combinators such that

$$\text{if } u_1 \equiv u_2, \text{ then } u_1^* \equiv u_2^*.$$

Hint: Define u^* by induction on the structure of u, using the compositional form of bracket abstraction considered in Exercise **3.5**. Show that the translation is itself compositional in that it commutes with substitution:

$$([u_2/x]u_1)^* = [u_2^*/x]u^*.$$

Then proceed by rule induction on rules (21.2) to show the required correctness condition.

Notes

1 It is debatable whether there are any scientific laws in Computer Science. In the opinion of the author Church's Law, which is usually called *Church's Thesis*, is a strong candidate for being a scientific law.

2 A property of untyped terms is *trivial* if it either holds for all untyped terms or never holds for any untyped term.

https://doi.org/10.1017/CBO9781316576892.023 Published online by Cambridge University Press