8 Function Definitions and Values

In the language **E**, we may perform calculations such as the doubling of a given expression, but we cannot express doubling as a concept in itself. To capture the pattern of doubling a number, we abstract away from the particular number being doubled using a *variable* to stand for a fixed, but unspecified, number, to express the doubling of an arbitrary number. Any particular instance of doubling may then be obtained by substituting a numeric expression for that variable. In general, an expression may involve many distinct variables, necessitating that we specify which of several possible variables is varying in a particular context, giving rise to a *function* of that variable.

In this chapter, we will consider two extensions of **E** with functions. The first, and perhaps most obvious, extension is by adding *function definitions* to the language. A function is defined by binding a name to an abt with a bound variable that serves as the argument of that function. A function is *applied* by substituting a particular expression (of suitable type) for the bound variable, obtaining an expression.

The domain and range of defined functions are limited to the types `nat` and `str`, because these are the only types of expression. Such functions are called *first-order* functions, in contrast to *higher-order functions*, which permit functions as arguments and results of other functions. Because the domain and range of a function are types, this requires that we introduce *function types* whose elements are functions. Consequently, we may form functions of *higher type*, those whose domain and range may themselves be function types.

8.1 First-Order Functions

The language **ED** extends **E** with function definitions and function applications as described by the following grammar:

Exp	e	::=	$\texttt{apply}\{f\}(e)$	$f(e)$	application
			$\texttt{fun}\{\tau_1;\tau_2\}(x_1.e_2;\, f.e)$	$\texttt{fun}\, f(x_1:\tau_1):\tau_2 = e_2\, \texttt{in}\, e$	definition

The expression $\texttt{fun}\{\tau_1;\tau_2\}(x_1.e_2;\, f.e)$ binds the function name f within e to the pattern $x_1.e_2$, which has argument x_1 and definition e_2. The domain and range of the function are, respectively, the types τ_1 and τ_2. The expression $\texttt{apply}\{f\}(e)$ instantiates the binding of f with the argument e.

https://doi.org/10.1017/CBO9781316576892.010 Published online by Cambridge University Press

The statics of **ED** defines two forms of judgment:

1. Expression typing, $e : \tau$, stating that e has type τ;
2. Function typing, $f(\tau_1) : \tau_2$, stating that f is a function with argument type τ_1 and result type τ_2.

The judgment $f(\tau_1) : \tau_2$ is called the *function header* of f; it specifies the domain type and the range type of a function.

The statics of **ED** is defined by the following rules:

$$\frac{\Gamma, x_1 : \tau_1 \vdash e_2 : \tau_2 \quad \Gamma, f(\tau_1) : \tau_2 \vdash e : \tau}{\Gamma \vdash \mathtt{fun}\{\tau_1; \tau_2\}(x_1.e_2; f.e) : \tau} \tag{8.1a}$$

$$\frac{\Gamma \vdash f(\tau_1) : \tau_2 \quad \Gamma \vdash e : \tau_1}{\Gamma \vdash \mathtt{apply}\{f\}(e) : \tau_2} \tag{8.1b}$$

Function substitution, written $[\![x.e/f]\!]e'$, is defined by induction on the structure of e' much like ordinary substitution. However, a function name f does not stand for an expression and can only occur in an application of the form $\mathtt{apply}\{f\}(e)$. Function substitution is defined by the following rule:

$$[\![x.e/f]\!]\mathtt{apply}\{f\}(e') = \mathtt{let}([\![x.e/f]\!]e'; x.e) \tag{8.2}$$

At application sites to f with argument e', function substitution yields a `let` expression that binds x to the result of expanding any further applications to f within e'.

Lemma 8.1. *If* $\Gamma, f(\tau_1) : \tau_2 \vdash e : \tau$ *and* $\Gamma, x_1 : \tau_1 \vdash e_2 : \tau_2$, *then* $\Gamma \vdash [\![x_1.e_2/f]\!]e : \tau$.

Proof By rule induction on the first premise, similarly to the proof of Lemma 4.4. □

The dynamics of **ED** is defined using function substitution:

$$\mathtt{fun}\{\tau_1; \tau_2\}(x_1.e_2; f.e) \longmapsto [\![x_1.e_2/f]\!]e \tag{8.3}$$

Because function substitution replaces all applications of f by appropriate `let` expressions, there is no need to give a rule for application expressions (essentially, they behave like variables that are replaced during evaluation, and not like a primitive operation of the language).

The safety of **ED** may, with some effort, be derived from the safety theorem for higher-order functions, which we discuss next.

8.2 Higher-Order Functions

The similarity between variable definitions and function definitions in **ED** is striking. Is it possible to combine them? The gap that must be bridged is the segregation of functions

https://doi.org/10.1017/CBO9781316576892.010 Published online by Cambridge University Press

from expressions. A function name f is bound to an abstractor $x.e$ specifying a pattern that is instantiated when f is applied. To reduce function definitions to ordinary definitions, we *reify* the abstractor into a form of expression, called a λ*-abstraction*, written $\mathtt{lam}\{\tau_1\}(x.e)$. Applications generalize to $\mathtt{ap}(e_1; e_2)$, where e_1 is an expression denoting a function, and not just a function name. λ-abstraction and application are the introduction and elimination forms for the *function type* $\mathtt{arr}(\tau_1; \tau_2)$, which classifies functions with domain τ_1 and range τ_2.

The language **EF** enriches **E** with function types, as specified by the following grammar:

Typ	τ	$::=$	$\mathtt{arr}(\tau_1; \tau_2)$	$\tau_1 \to \tau_2$	function
Exp	e	$::=$	$\mathtt{lam}\{\tau\}(x.e)$	$\lambda\,(x:\tau)\,e$	abstraction
			$\mathtt{ap}(e_1; e_2)$	$e_1(e_2)$	application

In **EF** functions are *first-class* in that they are a form of expression that can be used like any other. In particular, functions may be passed as arguments to, and returned as results from, other functions. For this reason, first-class functions are said to be *higher-order*, rather than *first-order*.

The statics of **EF** is given by extending rules (4.1) with the following rules:

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathtt{lam}\{\tau_1\}(x.e) : \mathtt{arr}(\tau_1; \tau_2)} \tag{8.4a}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{arr}(\tau_2; \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{ap}(e_1; e_2) : \tau} \tag{8.4b}$$

Lemma 8.2 (Inversion)**.** *Suppose that* $\Gamma \vdash e : \tau$.

1. *If* $e = \mathtt{lam}\{\tau_1\}(x.e_2)$*, then* $\tau = \mathtt{arr}(\tau_1; \tau_2)$ *and* $\Gamma, x : \tau_1 \vdash e_2 : \tau_2$.
2. *If* $e = \mathtt{ap}(e_1; e_2)$*, then there exists* τ_2 *such that* $\Gamma \vdash e_1 : \mathtt{arr}(\tau_2; \tau)$ *and* $\Gamma \vdash e_2 : \tau_2$.

Proof The proof proceeds by rule induction on the typing rules. Observe that for each rule, exactly one case applies and that the premises of the rule provide the required result. □

Lemma 8.3 (Substitution)**.** *If* $\Gamma, x : \tau \vdash e' : \tau'$*, and* $\Gamma \vdash e : \tau$*, then* $\Gamma \vdash [e/x]e' : \tau'$.

Proof By rule induction on the derivation of the first judgment. □

The dynamics of **EF** extends that of **E** with the following rules:

$$\frac{}{\mathtt{lam}\{\tau\}(x.e)\ \mathsf{val}} \tag{8.5a}$$

$$\frac{e_1 \longmapsto e_1'}{\mathtt{ap}(e_1; e_2) \longmapsto \mathtt{ap}(e_1'; e_2)} \tag{8.5b}$$

https://doi.org/10.1017/CBO9781316576892.010 Published online by Cambridge University Press

$$\left[\frac{e_1 \text{ val} \quad e_2 \longmapsto e_2'}{\mathtt{ap}(e_1; e_2) \longmapsto \mathtt{ap}(e_1; e_2')}\right] \tag{8.5c}$$

$$\frac{[e_2 \text{ val}]}{\mathtt{ap}(\mathtt{lam}\{\tau_2\}(x.e_1); e_2) \longmapsto [e_2/x]e_1} \tag{8.5d}$$

The bracketed rule and premise are included for a *call-by-value* interpretation of function application and excluded for a *call-by-name* interpretation.[1]

When functions are first class, there is no need for function declarations: simply replace the function declaration $\mathtt{fun}\, f(x_1 : \tau_1) : \tau_2 = e_2\, \mathtt{in}\, e$ by the definition $\mathtt{let}\, \lambda\,(x : \tau_1)\, e_2\, \mathtt{be}\, f\, \mathtt{in}\, e$, and replace second-class function application $f(e)$ by the first-class function application $f(e)$. Because λ-abstractions are values, it makes no difference whether the definition is evaluated by-value or by-name for this replacement to make sense. However, using ordinary definitions, we may, for example, give a name to a partially applied function, as in the following example:

$$\begin{array}{l}\mathtt{let}\ k\ \mathtt{be}\ \lambda\,(x_1 : \mathtt{num})\,\lambda\,(x_2 : \mathtt{num})\,x_1 \\ \mathtt{in\ let}\ kz\ \mathtt{be}\ k(0)\ \mathtt{in}\ kz(3) + kz(5).\end{array}$$

Without first-class functions, we cannot even form the function k, which returns a function as result when applied to its first argument.

Theorem 8.4 (Preservation)**.** *If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*

Proof The proof is by induction on rules (8.5), which define the dynamics of the language.

Consider rule (8.5d),

$$\frac{}{\mathtt{ap}(\mathtt{lam}\{\tau_2\}(x.e_1); e_2) \longmapsto [e_2/x]e_1}\ .$$

Suppose that $\mathtt{ap}(\mathtt{lam}\{\tau_2\}(x.e_1); e_2) : \tau_1$. By Lemma 8.2, we have $e_2 : \tau_2$ and $x : \tau_2 \vdash e_1 : \tau_1$, so by Lemma 8.3, $[e_2/x]e_1 : \tau_1$.

The other rules governing application are handled similarly. □

Lemma 8.5 (Canonical Forms)**.** *If $e : \mathtt{arr}(\tau_1; \tau_2)$ and e val, then $e = \lambda\,(x : \tau_1)\, e_2$ for some variable x and expression e_2 such that $x : \tau_1 \vdash e_2 : \tau_2$.*

Proof By induction on the typing rules, using the assumption e val. □

Theorem 8.6 (Progress)**.** *If $e : \tau$, then either e val, or there exists e' such that $e \longmapsto e'$.*

Proof The proof is by induction on rules (8.4). Note that because we consider only closed terms, there are no hypotheses on typing derivations.

Consider rule (8.4b) (under the by-name interpretation). By induction either e_1 val or $e_1 \longmapsto e_1'$. In the latter case, we have $\mathtt{ap}(e_1; e_2) \longmapsto \mathtt{ap}(e_1'; e_2)$. In the former case, we

https://doi.org/10.1017/CBO9781316576892.010 Published online by Cambridge University Press

have by Lemma 8.5 that $e_1 = \mathtt{lam}\{\tau_2\}(x.e)$ for some x and e. But then $\mathtt{ap}(e_1; e_2) \longmapsto [e_2/x]e$. □

8.3 Evaluation Dynamics and Definitional Equality

An inductive definition of the evaluation judgment $e \Downarrow v$ for **EF** is given by the following rules:

$$\frac{}{\mathtt{lam}\{\tau\}(x.e) \Downarrow \mathtt{lam}\{\tau\}(x.e)} \tag{8.6a}$$

$$\frac{e_1 \Downarrow \mathtt{lam}\{\tau\}(x.e) \quad [e_2/x]e \Downarrow v}{\mathtt{ap}(e_1; e_2) \Downarrow v} \tag{8.6b}$$

It is easy to check that if $e \Downarrow v$, then v val, and that if e val, then $e \Downarrow e$.

Theorem 8.7. *$e \Downarrow v$ iff $e \longmapsto^* v$ and v val.*

Proof In the forward direction, we proceed by rule induction on rules (8.6), following along similar lines as the proof of Theorem 7.2.

In the reverse direction, we proceed by rule induction on rules (5.1). The proof relies on an analog of Lemma 7.4, which states that evaluation is closed under converse execution, which is proved by induction on rules (8.5). □

Definitional equality for the call-by-name dynamics of **EF** is defined by extension of rules (5.10).

$$\frac{}{\Gamma \vdash \mathtt{ap}(\mathtt{lam}\{\tau\}(x.e_2); e_1) \equiv [e_1/x]e_2 : \tau_2} \tag{8.7a}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \tau_2 \to \tau \quad \Gamma \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash \mathtt{ap}(e_1; e_2) \equiv \mathtt{ap}(e_1'; e_2') : \tau} \tag{8.7b}$$

$$\frac{\Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash \mathtt{lam}\{\tau_1\}(x.e_2) \equiv \mathtt{lam}\{\tau_1\}(x.e_2') : \tau_1 \to \tau_2} \tag{8.7c}$$

Definitional equality for call-by-value requires a bit more machinery. The main idea is to restrict rule (8.7a) to require that the argument be a value. In addition, values must be expanded to include variables, because in call-by-value, the argument variable of a function stands for the value of its argument. The call-by-value definitional equality judgment takes the form

$$\Gamma \vdash e_1 \equiv e_2 : \tau,$$

where Γ consists of paired hypotheses $x : \tau, x$ val stating, for each variable x in scope, its type and that it is a value. We write $\Gamma \vdash e$ val to show that e is a value under these hypotheses, so that $x : \tau, x$ val $\vdash x$ val.

https://doi.org/10.1017/CBO9781316576892.010 Published online by Cambridge University Press

8.4 Dynamic Scope

The dynamics of function application given by rules (8.5) is defined only for expressions without free variables. When a function is applied, the argument is substituted for the argument variable, ensuring that the result remains closed. Moreover, because substitution of closed expressions can never incur capture, the scopes of variables are not disturbed by the dynamics, ensuring that the principles of binding and scope described in Chapter 1 are respected. This treatment of variables is called *static scoping*, or *static binding*, to contrast it with an alternative approach that we now describe.

Another approach, called *dynamic scoping*, or *dynamic binding*, is sometimes advocated as an alternative to static binding. The crucial difference is that with dynamic scoping the principle of identification of abt's up to renaming of bound variables is *denied*. Consequently, capture-avoiding substitution is not available. Instead, evaluation is defined for open terms, with the bindings of free variables provided by an *environment* mapping variable names to (possibly open) values. The binding of a variable is determined as late as possible, at the point where the variable is evaluated, rather than where it is bound. If the environment does not provide a binding for a variable, evaluation is aborted with a run-time error.

For first-order functions, dynamic and static scoping coincide, but in the higher-order case, the two approaches diverge. For example, there is no difference between static and dynamic scope when it comes to evaluation of an expression such as $(\lambda\,(x : \mathtt{num})\,x + 7)(42)$. Whether 42 is substituted for x in the body of the function before evaluation, or the body is evaluated in the presence of the binding of x to 42, the outcome is the same.

In the higher-order case, the equivalence of static and dynamic scope breaks down. For example, consider the expression

$$e \triangleq (\lambda\,(x : \mathtt{num})\,\lambda\,(y : \mathtt{num})\,x + y)(42).$$

With static scoping e evaluates to the closed value $v \triangleq \lambda\,(y : \mathtt{num})\,42 + y$, which, if applied, would add 42 to its argument. It makes no difference how the bound variable x is chosen, the outcome will always be the same. With dynamic scoping, e evaluates to the open value $v' \triangleq \lambda\,(y : \mathtt{num})\,x + y$ in which the variable x occurs free. When this expression is evaluated, the variable x is bound to 42, but this is irrelevant because the binding is not needed to evaluate the λ-abstraction. The binding of x is not retrieved until such time as v' is applied to an argument, at which point the binding for x in force at that time is retrieved, and not the one in force at the point where e is evaluated.

Therein lies the difference. For example, consider the expression

$$e' \triangleq (\lambda\,(f : \mathtt{num} \to \mathtt{num})\,(\lambda\,(x : \mathtt{num})\,f(0))(7))(e).$$

When evaluated using dynamic scope, the value of e' is 7, whereas its value is 42 under static scope. The discrepancy can be traced to the re-binding of x to 7 before the value of e, namely v', is applied to 0, altering the outcome.

Dynamic scope violates the basic principle that variables are given meaning by capture-avoiding substitution as defined in Chapter 1. Violating this principle has at least two

https://doi.org/10.1017/CBO9781316576892.010 Published online by Cambridge University Press

undesirable consequences. One is that the names of bound variables matter, in contrast to static scope which obeys the identification principle. For example, had the innermost λ-abstraction of e' bound the variable y, rather than x, then its value would have been 42, rather than 7. Thus, one component of a program may be sensitive to the names of bound variables chosen in another component, a clear violation of modular decomposition.

Another problem is that dynamic scope is not, in general, type-safe. For example, consider the expression

$$e' \triangleq (\lambda\,(f : \mathtt{num} \to \mathtt{num})\,(\lambda\,(x : \mathtt{str})\; f(\text{"}zero\text{"}))(7))(e).$$

Under dynamic scoping this expression gets stuck attempting to evaluate $x + y$ with x bound to the string "*zero*," and no further progress can be made. For this reason dynamic scope is only ever advocated for so-called dynamically typed languages, which replace static consistency checks by dynamic consistency checks to ensure a weak form of progress. Compile-time errors are thereby transformed into run-time errors.

(For more on dynamic typing, see Chapter 22, and for more on dynamic scope, see Chapter 32.)

8.5 Notes

Nearly all programming languages provide some form of function definition mechanism of the kind illustrated here. The main point of the present account is to demonstrate that a more natural, and more powerful, approach is to separate the generic concept of a definition from the specific concept of a function. Function types codify the general notion in a systematic way that encompasses function definitions as a special case, and moreover, admits passing functions as arguments and returning them as results without special provision. The essential contribution of Church's λ-calculus (Church, 1941) was to take functions as primary, and to show that nothing more is needed to get a fully expressive programming language.

Exercises

8.1. Formulate an environmental evaluation dynamics (see Exercise **7.5**) for **ED**. *Hint*: Introduce a new form of judgment for evaluation of function identifiers.

8.2. Consider an environmental dynamics for **EF**, which includes higher-order functions. What difficulties arise? Can you think of a way to evade these difficulties? *Hint*: One approach is to "substitute away" all free variables in a λ-abstraction at the point at which it is evaluated. The second is to "freeze" the values of each of the free variables in a λ-abstraction, and to "thaw" them when such a function is applied. What problems arise in each case?

https://doi.org/10.1017/CBO9781316576892.010 Published online by Cambridge University Press

Note

1 Although the term “call-by-value” is accurately descriptive, the origin of the term “call-by-name” remains shrouded in mystery.

https://doi.org/10.1017/CBO9781316576892.010 Published online by Cambridge University Press