2 The Simply-Typed λ-Calculus

The well-deserved starting point for this discussion of the semantics of programming languages is the simply-typed λ-calculus. The calculus dates back to before the advent of modern higher-order programming languages. It was first studied in the 1930's by Church and others working on the foundations of mathematics. Their results had clear computational significance and were related to the original ideas on which the concept of a computer was based. However, the relevance of the simply-typed calculus (as opposed to the untyped calculus which is discussed later) was not fully appreciated by computer scientists until the 1960's, when its usefulness as the basis of a higher-order language was studied and its potential as a specification language was noted. The calculus can now be found as a fragment of many well-known programming languages, and it is the bedrock of work in the mathematical semantics of programming languages.

The simply-typed λ-calculus is a fundamental and powerful theory of *application and abstraction* governed by a basic theory of *types.* A type can be viewed as a property of an expression that describes how the expression can be used. We say that an expression (program) M *has type* $s \to t$ if it can be applied to an expression of type s to yield an expression of type t. As trivial as it sounds, this rule is fundamental to all of our subsequent discussion of types. The expressions of the calculus are also taken to satisfy certain rules. The most important of these is called the *β-rule,* and it says that the application of a function to an argument is equal to the result of substituting the argument for the formal parameters used in defining the function. This sounds much simpler than it really is, and, indeed, the earliest uses of the λ-calculus in programming languages were subject to the problems of clarifying many nuances about what this statement actually means. Note, in particular, that understanding the β-rule involves understanding substitution.

In this chapter we look at the basic syntax and semantics of the simply-typed λ-calculus in an essentially non-computational setting. Its dynamic or operational meaning(s) will be discussed later. In the meantime we touch on several topics that will be fundamental to everything discussed later. After giving the syntax of expressions, the typing rules, and the equational rules for the calculus, the simplest example of a model is given. The *general definition* of a model is then defined rigorously, using the concept of a *type frame,* and other examples of models are given. Using the concept of a *partial homomorphism* between type frames, it is shown that our model using sets and functions is *complete* for the equational rules of the calculus.

2.1 Syntax of λ-Terms

To understand the syntax of the λ-calculus, it is essential to distinguish between three separate notions:

1. the concrete syntax of expressions used to denote term trees,
2. the term trees generated by the context-free grammar, and
3. equivalence classes of term trees modulo renaming of bound variables.

The last of these are the λ-terms, and they are our primary interest. To understand them, it is essential that the parsing conventions that relate concrete syntax to term trees are fully stated and that the equivalence relation on term trees is precisely defined. At one time or another it will be necessary to direct our attention to each of the three levels of description. Let us begin with the context-free grammar for types and terms; it is given in BNF as follows:

$$\begin{array}{rcl} x & \in & \text{Variable} \\ t & ::= & \mathbf{o} \mid t \to t \\ M & ::= & x \mid \lambda x : t.\ M \mid MM \end{array}$$

where Variable is the primitive syntax class of *variables.*[1] The expressions in the syntax class over which t ranges are called *types,* and those in the class over which M ranges are called *term trees.* In a later chapter, we expand our horizons to admit the possibility that there are variables in the syntax class of types as well as that of term trees; the adjective 'simply-typed' is meant to indicate that our calculus does not include this feature. In general, I use letters from the end of the alphabet such as x, y, z and such letters with subscripts and superscripts as in x', x_1, x_2 to range over variables, but it is also handy to use letters such as f, g for variables in some cases. Types are generally written using letters r, s, and t. Term trees are generally written with letters L, M, N. Such letters with superscripts and subscripts may also be used when convenient.

The type $\mathbf{o}$ is called the *ground* type, and types $s \to t$ are called *higher* types. Term trees of the form $\lambda x : t.\ M$ are called *abstractions,* and those of the form MN are called *applications.* Parentheses are used to indicate how an expression is parsed. For example, without a convention about parsing, the expression $\mathbf{o} \to \mathbf{o} \to \mathbf{o}$ is ambiguous and could be parsed as $(\mathbf{o} \to \mathbf{o}) \to \mathbf{o}$ or $\mathbf{o} \to (\mathbf{o} \to \mathbf{o})$. Similarly, without a convention, $\lambda x : s.\ \lambda y : t.\ xyz$ could be parsed as $(\lambda x : s.\ \lambda y : t.\ x)(yz)$ or $(\lambda x : s.\ \lambda y : t.\ (xy))z$ or as any one of a variety of other term trees. These two term trees are pictured in Figure 2.1

[1]Sometimes a distinction is made between a variable and an *identifier.* No such distinction will be made in this book and from now on the term 'variable' will generally be used in place of 'identifier'.

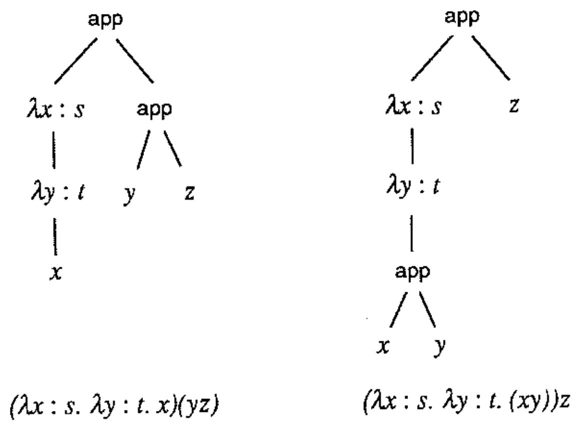

Figure 2.1
Examples of Term Trees

with application nodes labeled by 'appl' and abstraction nodes labeled by the variable and type.

To keep the number of parentheses at a minimum we employ several standard parsing conventions. For types, the *association of the operator* $\rightarrow$ *is to the right:* thus $\mathbf{o} \rightarrow \mathbf{o} \rightarrow \mathbf{o}$ parses as $\mathbf{o} \rightarrow (\mathbf{o} \rightarrow \mathbf{o})$. Dually, *application operations associate to the left:* an application LMN should be parsed as $(LM)N$. So, the expression xyz unambiguously parses as $(xy)z$. If we wish to write the expression that applies x to the result of applying y to z, it is rendered as $x(yz)$. Moreover *application binds more tightly than abstraction:* an expression $\lambda x : t.\ MN$ should be parsed as $\lambda x : t.\ (MN)$. Hence, the expression $\lambda x : s.\ \lambda y : t.\ xyz$ unambiguously parses as $\lambda x : s.\ \lambda y : t.\ ((xy)z)$. Superfluous parentheses can be sprinkled into an expression at will to emphasize grouping. There is no distinction between M and (M), and it is common to surround the operand of an application $M(N)$ with parentheses to mimic the mathematical notation $f(x)$ for a function applied to an argument.[2] It is important to understand the role of meta-variables such as M, N, and t in parsing an expression involving such letters. For example, if M and N are the expressions $\lambda x : \mathbf{o}.\ x$ and y respectively, then the expression MN is $(\lambda x : \mathbf{o}.\ x)y$ and *not* the expression $\lambda x : \mathbf{o}.\ (xy)$, which one might get by concatenating $\lambda x : \mathbf{o}.\ x$ with y and then applying the parsing conventions. A simple strategy for avoiding any error is

[2]This is especially worth noting for those familiar with Lisp-like syntax where there is a big difference between f and (f) for an identifier f.

to replace MN by $(M)(N)$ before expanding M and N.

An *occurrence* of a variable in an expression is simply a point in the expression where the variable appears. For example, the underscore indicates:

1. the first occurrence of x in $\lambda \underline{x} : \mathbf{o}.\ (yx)x$
2. the second occurrence of x in $\lambda x : \mathbf{o}.\ (y\underline{x})x$
3. the third occurrence of x in $\lambda x : \mathbf{o}.\ (yx)\underline{x}$

Of course, occurrences can also be described using the tree structure of a term tree by indicating the position of the node in the tree where the variable lies. However, counting occurrences from left to right in the concrete syntax denoting the term tree is a simple approach that is sufficient for our purposes. In an abstraction $\lambda x : t.\ M$, the first occurrence of x is called the *binding occurrence* of the abstraction. The variable x is called the *formal parameter*. The term tree M is called the *body* of the abstraction, and t is the *type tag* of the binding occurrence. A period follows the type tag in the concrete syntax and separates the tag from the body of the abstraction. In an application $M(N)$, the term tree M is the *operator* and N is the *operand* or *argument*.

We may extend the notion of an occurrence to apply to arbitrary term trees as well as variables. For example, there are two occurrences of the term tree $\lambda x : s.\ x$ in the following expression:

$$(\lambda y : t.\ \underbrace{\lambda x : s.\ x}_{\text{first}})(\underbrace{\lambda x : s.\ x}_{\text{second}})$$

Such an occurrence of a term tree within another term tree is called a *subterm*. Care must be taken not to confuse the occurrence of a term tree as a subterm with a *substring* that would denote the term tree in concrete syntax. For example, there is only *one* occurrence of the term tree $\lambda x : s.\ x$ as a subterm of

$$\lambda y : t.\ \lambda x : s.\ x(\underbrace{\lambda x : s.\ x})$$

as indicated by the bracket.

An occurrence of a variable x in a term tree M is *free* if one of the following conditions holds:

- M is x, or
- M has the form $\lambda y : t.\ N$, where y is different from x and the occurrence of x in the subterm N is free, or
- M has the form $L(N)$ and the occurrence of x is free in L or in N.

Let M be a term tree and suppose we are given a binding occurrence of a variable x in M. Suppose, in particular, that the binding occurrence of x in question is in a subterm $\lambda x : t.\ N$ of M. An occurrence of x in M is said to be in the *scope* of this binding if it is free in N. Free occurrences of x in N are said to be *bound* to the binding occurrence of x in the abstraction. The set $\mathrm{Fv}(M)$ of variables having free occurrences in M is defined inductively as follows:

- $\mathrm{Fv}(x) = \{x\}$
- $\mathrm{Fv}(\lambda x : t.\ N) = \mathrm{Fv}(N) - \{x\}$
- $\mathrm{Fv}(L(N)) = \mathrm{Fv}(L) \cup \mathrm{Fv}(N)$

The treatment of bound variables is one of the most subtle points in the study of the λ-calculus or, indeed, in almost any of the areas in computer science and logic where the concept arises. In general, we will want to treat bound variables as 'nameless'. In this chapter, for example, we will not wish to distinguish between the term trees $\lambda x : \mathbf{o}.\ x$ and $\lambda y : \mathbf{o}.\ y$ in which the names of the bound variable are different. However, in a later chapter we will be interested in the idea of substituting a term tree into another term tree in such a way that the names of bound variables *are* essential. For example, if $\lambda x : \mathbf{o}.\ yx$ is viewed as the text of a program, then there is a big difference between putting it in place of the brackets in the expression $\lambda y : \mathbf{o} \to \mathbf{o}.\ y\{\ \}$ versus putting it in place of the brackets in the expression $\lambda x : \mathbf{o} \to \mathbf{o}.\ x\{\ \}$ since the first substitution creates a new binding whereas the second does not. A reasonable theory of program transformations dealing with the substitutability of code fragments should allow for this possibility.

To study this notion further and prepare ourselves for the definitions needed to explicate this dichotomy precisely, we define the *context substitution* $\{M/x\}N$ of a term tree M for a variable x in a term tree N inductively as follows:

- if N is the variable x, then $\{M/x\}N$ is M.
- if N is a variable y different from x, then $\{M/x\}N$ is y.
- if N is an application $N_1(N_2)$, then $\{M/x\}N$ is $(\{M/x\}N_1)(\{M/x\}N_2)$.
- if N is an abstraction $\lambda x : t.\ L$ where the bound variable is x, then $\{M/x\}N$ is N.
- if N is an abstraction $\lambda y : t.\ L$ where the bound variable y is different from x, then $\{M/x\}N$ is $\lambda y : t.\ \{M/x\}L$.

For example, $\{\lambda x : \mathbf{o}.\ yx/z\}\lambda y : \mathbf{o}.\ yz$ is the term tree $\lambda y : \mathbf{o}.\ y(\lambda x : \mathbf{o}.\ yx)$ whereas $\{\lambda x : \mathbf{o}.\ yx/z\}\lambda x : \mathbf{o}.\ xz$ is $\lambda x : \mathbf{o}.\ x(\lambda x : \mathbf{o}.\ yx)$. This notion of substitution *allows* the possibility that an occurrence of a variable that is free in M becomes bound in its occurrence in $\{M/x\}N$.

In most of the literature on the λ-calculus, this capture of free variables under substitution is considered undesirable, and substitution is defined with conditions that avoid the problem. However, the matter has no trivial resolution since it arises from the deep fact that there is a distinction between the textual representation of terms of the λ-calculus, the trees generated by our grammar for such terms, and the underlying structure that generally interests us. In expositions of the λ-calculus, the matter is typically swept under the carpet in one way or another. However, it arises inevitably in any attempt to automate or implement the λ-calculus as a formal system. I will not attempt to discuss the techniques that are used to deal with these implementation problems here since they are not immediately pertinent to the needs of this chapter; instead, I will simply distinguish between the term trees arising from the grammar above and equivalence classes of such terms modulo renaming of bound variables.

Definition: A *relation* is a triple consisting of a pair of sets X, Y called the *domain* and *codomain* (or *range*) of the relation and a set R of pairs (x, y) where $x \in X$ and $y \in Y$. We write $R : X \to Y$ to indicate that R is a relation with domain X and codomain Y. It is common to use an infix notation for relations by writing $x \; R \; y$ if $(x, y) \in R$.

Given a set X, a *binary relation on* X is a relation $R : X \to X$. Such a relation is said to be an *equivalence* if it satisfies the following three properties for any $x, y, z \in X$:

1. Reflexive: $x \; R \; x$;
2. Symmetric: $x \; R \; y$ implies $x \; R \; y$;
3. Transitive: $x \; R \; y$ and $y \; R \; z$ implies $x \; R \; z$. □

Definition: The *α-equivalence* of term trees is defined to be the least relation $\equiv$ between term trees such that

- $x \equiv x$
- $MN \equiv M'N'$ if $M \equiv M'$ and $N \equiv N'$
- $\lambda x : t.\ M \equiv \lambda y : t.\ N$ if $\{z/x\}M \equiv \{z/y\}N$ where z is a variable that has no occurrence in M or N. □

2.1 Lemma. *The relation $\equiv$ is an equivalence.* □

The proof is left as an exercise. Equivalence classes of term trees modulo the relation $\equiv$ are called λ-*terms* or, more simply, *terms*. The equivalence class of term trees determined by a term tree M is called its *α-equivalence class*. In general, the same letters (L, M, N, *etc.*) are used for both λ-terms and term trees, with the understanding that a term is given as the α-equivalence class of a term tree representative. However, for the moment, let us be especially precise and write M^* for the α-equivalence class of M. Substitution on λ-terms is defined to respect these equivalence classes, and our notation for it is similar to the one we used for term trees.

Definition: We define the substitution $[M^*/x]N^*$ of M^* for x in N^* to be the α-equivalence class of $\{M/x\}L$ where $L \equiv N$ and no free variable of M has a binding occurrence in L. □

For example, if M is $\lambda x : \mathbf{o}.\ yx$ and N is $\lambda y : \mathbf{o}.\ yz$, then $[M^*/z]N^*$ is $(\lambda u : \mathbf{o}.\ u(\lambda x : \mathbf{o}.\ yx))^*$ where u is some new variable. What if we had chosen another variable besides u? If the new choice is v, say, this would give us a different term tree, but

$$\lambda u : \mathbf{o}.\ u(\lambda x : \mathbf{o}.\ yx) \equiv \lambda v : \mathbf{o}.\ v(\lambda x : \mathbf{o}.\ yx)$$

so this choice makes no difference in determining the equivalence class. To be fully precise we need to prove a lemma that the substitution operation is well-defined on α-equivalence classes. The proof is left for the reader.

In the future we will work with λ-terms as equivalence classes of term trees without explicitly mentioning term trees. To rid our discussion of many tedious repetitions of assumptions about the names of bound variables, it is helpful to use a convention about the choice of the representative of an α-equivalence class.

Notation: (Bound Variable Naming Convention.) When a term tree representative of an α-equivalence class is chosen, the name of the bound variable of the representative is distinct from the names of free variables in other terms being discussed. □

When we encounter the need to deal explicitly with the names of bound variables, then term trees will be distinguished from λ-terms explicitly.

Although we are not concerned with the possibility of bound variables at the level of types in this chapter, this will become an issue later. So, in anticipation of this expansion, we write $s \equiv t$ to indicate that the types s and t generated from our grammar for types are the same. For instance, $\mathbf{o} \to \mathbf{o} \to \mathbf{o} \equiv \mathbf{o} \to (\mathbf{o} \to \mathbf{o})$, but $\mathbf{o} \to \mathbf{o} \to \mathbf{o} \not\equiv (\mathbf{o} \to \mathbf{o}) \to \mathbf{o}$.

Exercises.

2.1 Assuming that there are no parsing conventions, list all of the ways in which the expression $\lambda x : s.\ \lambda y : t.\ xyz$ could potentially be parsed. For each possible parsing, indicate the relationship between all binding and bound occurrences of variables.

2.2 Prove that any type t can be written in the form $t_1 \to t_2 \to \cdots \to t_n \to o$ for some $n \geq 0$ and type expressions $t_1, \ldots t_n$.

Table 2.1
Typing Rules for the Simply-Typed λ-Calculus

[Proj]	$H, x : t, H' \vdash x : t$
[Abs]	$\dfrac{H,\ x : s \vdash M : t}{H \vdash \lambda x : s.\ M : s \to t}$
[Appl]	$\dfrac{H \vdash M : s \to t \qquad H \vdash N : s}{H \vdash M(N) : t}$

2.2 Rules

There are two systems of rules describing the simply-typed λ-calculus. The first of these determines which of the terms described in the previous section are to be viewed as *well-typed*. These are the terms to which we will assign a meaning in our semantic model. The second set of rules of the calculus forms its *equational theory*, which consists of a set of rules for proving equalities between λ-terms.

Typing rules.

A *type assignment* is a list $H \equiv x_1 : t_1, \ldots, x_n : t_n$ of pairs of variables and types such that the variables x_i are distinct. The empty type assignment $\emptyset$ is the degenerate case in which there are no pairs. I write $x : t \in H$ if x is x_i and t is t_i for some i. In this case it is said that x *occurs* in H, and this may be abbreviated by writing $x \in H$. If $x : t \in H$, then define $H(x)$ to be the type t.

A *typing judgement* is a triple consisting of a type assignment H, a term M, and a type t such that all of the free variables of M appear in H. We will write this relation between H, M, and t in the form $H \vdash M : t$ and read it as 'in the assignment H, the term M has type t'. It is defined to be the least relation satisfying the axiom and two rules in Table 2.1. Here are some comments on the rules:

- The projection rule [Proj] asserts, in effect, that $H \vdash x : H(x)$ provided $x \in H$.
- The application rule [Appl] says that, for a type assignment H, a term of the form $M(N)$ has a type t if there is a type s such that the operand N has type s and the operator M has type $s \to t$.
- The abstraction rule [Abs] is the most subtle of the typing rules. It says that a term of the form $\lambda x : s.\ M$ has the type $s \to t$ in type assignment H if M has type t in type assignment $H, x : s$. Note, however, the relationship between the conclusion

of the rule, where x represents a bound variable on the right side of the $\vdash$, and its hypothesis, where x is a free variable in M. It is essential that the bound variable in the term tree representative of the conclusion not appear in H since otherwise $H, x : s$ would not be a well-formed type assignment.

A demonstration of $H \vdash M : t$ from these rules is called a *typing derivation.* Such derivations are trees with nodes labeled by typing judgements. As an example, consider how it can be shown that

$$\vdash \lambda x : \mathbf{o}.\ \lambda f : \mathbf{o} \to \mathbf{o}.\ f(x) : \mathbf{o} \to (\mathbf{o} \to \mathbf{o}) \to \mathbf{o}$$

To derive the desired conclusion, begin by noting that the hypotheses of the following instance of the application rule both follow from projection:

$$\frac{x : \mathbf{o}, f : \mathbf{o} \to \mathbf{o} \vdash f : \mathbf{o} \to \mathbf{o} \qquad x : \mathbf{o}, f : \mathbf{o} \to \mathbf{o} \vdash x : \mathbf{o}}{x : \mathbf{o}, f : \mathbf{o} \to \mathbf{o} \vdash f(x) : \mathbf{o}}$$

and two instances of the abstraction rule can be used to complete the proof:

$$\frac{\dfrac{x : \mathbf{o}, f : \mathbf{o} \to \mathbf{o} \vdash f(x) : \mathbf{o}}{x : \mathbf{o} \vdash \lambda f : \mathbf{o} \to \mathbf{o}.\ f(x) : (\mathbf{o} \to \mathbf{o}) \to \mathbf{o}}}{\vdash \lambda x : \mathbf{o}.\ \lambda f : \mathbf{o} \to \mathbf{o}.\ f(x) : \mathbf{o} \to (\mathbf{o} \to \mathbf{o}) \to \mathbf{o}}$$

Large proofs of this kind can become cumbersome to write as a tree on the page; to get a tabular representation that can be written as three columns, one can list the addresses of the nodes of the tree in postfix order with the labeling judgement written next to the address. Here is an example in which the nodes are also labeled with the rule used:

000	$x : \mathbf{o}, f : \mathbf{o} \to \mathbf{o} \vdash f : \mathbf{o} \to \mathbf{o}$	[Proj]
001	$x : \mathbf{o}, f : \mathbf{o} \to \mathbf{o} \vdash x : \mathbf{o}$	[Proj]
00	$x : \mathbf{o}, f : \mathbf{o} \to \mathbf{o} \vdash f(x) : \mathbf{o}$	[Appl]
0	$x : \mathbf{o} \vdash \lambda f : \mathbf{o} \to \mathbf{o}.\ f(x) : (\mathbf{o} \to \mathbf{o}) \to \mathbf{o}$	[Abs]
	$\vdash \lambda x : \mathbf{o}.\ \lambda f : \mathbf{o} \to \mathbf{o}.\ f(x) : \mathbf{o} \to (\mathbf{o} \to \mathbf{o}) \to \mathbf{o}$	[Abs]

In this addressing scheme, if b is any sequence of zeroes and ones, $b0$ and $b1$ are the first and second sons respectively of the node addressed by b. For example, the judgement at node 00 in this proof follows by the application rule from hypotheses 000 and 001. Each of these hypotheses is an instance of the projection axiom. In some cases a rule has only one hypothesis, and in such cases it is addressed as a first son; for instance, 0 follows from 00 by abstraction. The conclusion is addressed by the empty sequence and appears as the last line of the derivation.

To illustrate the care that must be taken in dealing with bound variables in the abstraction rule, let $t \equiv \mathbf{o} \to (\mathbf{o} \to \mathbf{o}) \to \mathbf{o}$ and consider the term

$$M \equiv \lambda x : \mathbf{o}.\ \lambda y : t.\ yx(\lambda x : \mathbf{o}.\ x).$$

This term has type $\mathbf{o} \to t \to \mathbf{o}$. To prove this, it is essential to work with a term tree representative that uses a different name for one of the bound occurrences of x. Here is a derivation:

0000	$x : \mathbf{o},\ y : t \vdash y : t$	[Proj]
0001	$x : \mathbf{o},\ y : t \vdash x : \mathbf{o}$	[Proj]
000	$x : \mathbf{o},\ y : t \vdash yx : (\mathbf{o} \to \mathbf{o}) \to \mathbf{o}$	[Appl]
0010	$x : \mathbf{o},\ y : t,\ z : \mathbf{o} \vdash z : \mathbf{o}$	[Proj]
001	$x : \mathbf{o},\ y : t \vdash \lambda z : \mathbf{o}.\ z : \mathbf{o} \to \mathbf{o}$	[Abs]
00	$x : \mathbf{o},\ y : t \vdash yx(\lambda z : \mathbf{o}.\ z) : \mathbf{o}$	[Appl]
0	$x : \mathbf{o} \vdash \lambda y : t.\ yx(\lambda z : \mathbf{o}.\ z) : t \to \mathbf{o}$	[Abs]
	$\vdash \lambda x : \mathbf{o}.\ \lambda y : t.\ yx(\lambda z : \mathbf{o}.\ z) : \mathbf{o} \to t \to \mathbf{o}$	[Abs]

Since the term tree representative used for the term M above is α-equivalent to the last term in this derivation tree, we conclude that M has type $\mathbf{o}$.

In general, we will only be interested in terms M and type assignments H such that $H \vdash M : t$ for some type t. Arbitrary terms (including terms that do not have a type) are sometimes called 'raw terms' since they are terms that do not satisfy any such relation. For example, $\lambda x : \mathbf{o}.\ x(x)$ fails to have a type in any type assignment. But if a term has a type in a given assignment, then that type is uniquely determined. To prove this, and many other properties of the λ-calculus, one can use a technique known as *structural induction*. Essentially, this is an induction on the height of the parse tree of the term; a property of terms M is proved by first showing that it holds if M is a variable and then showing that the property is satisfied by M if it is satisfied by each of its subterms.

2.2 Lemma. *If $H \vdash M : t$ and x does not appear in H, then x is not free in M.*

Proof: If M is a variable y, then it cannot be x since $H \vdash M : t$ must follow from projection, so $y \in H$. Suppose the lemma is known for subterms of M; that is, suppose that, for any subterm M' of M, if $H' \vdash N : t'$ and x' is a variable that does not appear in H', then x' is not free in M'. Now the goal is to prove that the lemma holds for M itself.

If M is an abstraction, then M must have the form $\lambda y : r.\ M'$ where $H,\ y : r \vdash M' : s$ and $t \equiv r \to s$. If it happens that this y is the same as x, then the desired conclusion is immediate since x is not free in $\lambda x : r.\ M'$. If, on the other hand, y is different from

x, then x does not appear in $H,\ y : r$ so the fact that the lemma is assumed, by the inductive hypothesis, to hold for subterms of M means that it holds for M'. Hence x is not free in M' and therefore not free in M.

If M is an application, then it has the form $L(N)$. Since the application rule is the only way to prove that this term has a type, we must have $H \vdash L : s \to t$ and $H \vdash N : s$ for some type s. The inductive hypothesis says that the lemma is true for these two subterms of M, so x is free in neither L nor N. Therefore, it is not free in M. □

A similar argument can be used to prove the following:

2.3 Lemma. *If $H \vdash \lambda x : s.\ M : t$, then $H,\ y : s \vdash [y/x]M : t$ for any variable y that does not appear in H.* □

The proof is left as an exercise.

2.4 Lemma. *If $H \vdash M : s$ and $H \vdash M : t$, then $s \equiv t$.*

Proof: The proof is by induction on the structure of M. If $M \equiv x$ for a variable x then $s \equiv H(x) \equiv t$.

If $M \equiv \lambda x : u.\ M'$, then $s \equiv u \to s'$ where $H,\ x : u \vdash M' : s'$ and also $t \equiv u \to t'$ where $H,\ x : u \vdash M' : t'$ (by Lemma 2.3 and a suitable choice of x). Since M' is a subterm of M, we may conclude that it satisfies the lemma and that $s' \equiv t'$. But this immediately implies that $s \equiv t$.

If $M \equiv L(N)$, then $H \vdash L : u \to s$ for some u. Similarly, there is a type v such that $H \vdash L : v \to t$. Since L is a subterm of M, we must have $u \to s \equiv v \to t$ and therefore $s \equiv t$. □

Type tags are placed on bound variables in abstractions just to make Lemma 2.4 true. If we try to simplify our notation by allowing terms of the form $\lambda x.\ M$ and a typing rule of the form

$$[\text{Abs}]^{-} \qquad \frac{H,\ x : s \vdash M : t}{H \vdash \lambda x.\ M : s \to t},$$

then Lemma 2.4 would *fail*. For example, we would then have

$$\vdash \lambda x.\ x : \mathbf{o} \to \mathbf{o}$$

as well as

$$\vdash \lambda x.\ x : (\mathbf{o} \to \mathbf{o}) \to (\mathbf{o} \to \mathbf{o}).$$

Most of the calculi introduced in this book have a syntax with enough type tags to ensure that an appropriate version of Lemma 2.4 holds.

There are several basic properties of type assignments that can be proved by structural induction.

2.5 Lemma. *If $H,\ x : r,\ y : s,\ H' \vdash M : t$, then $H,\ y : s,\ x : r,\ H' \vdash M : t$.*

Proof: The proof proceeds by induction on the structure of the term M. Assume, as the inductive hypothesis, that the lemma holds for any subterm of M. Let $H_0 \equiv H,\ x : r,\ y : s,\ H'$ and $H_1 \equiv H,\ y : s,\ x : r,\ H'$.

Case $M \equiv x$. Then $r \equiv t$, and the projection axiom applies. A similar argument applies when $M \equiv y$.

Case $M \equiv z$ where $z \not\equiv x$ and $z \not\equiv y$. Then $z : t \in H$ or $z : t \in H'$, and the projection axiom applies.

Case $M \equiv \lambda z : u.\ N$. Then $t \equiv u \to v$ for some type v. Since $H_0 \vdash M : u \to v$, we must have $H_0, z : u \vdash N : v$ so $H_1, z : u \vdash N : v$ follows from the inductive hypothesis. Hence, by the abstraction rule, $H_1 \vdash M : t$ as desired.

Case $M \equiv L(N)$. In this case, there is some type u such that $H_0 \vdash L : u \to t$ and $H_0 \vdash N : u$ since this is the only way $H_0 \vdash M : t$ could have been derived. By the inductive hypothesis, we may conclude that $H_1 \vdash L : u \to t$ and $H_1 \vdash N : u$. By the application rule, we must therefore have $H_1 \vdash M : t$ as desired. □

The proof of the lemma may equivalently be viewed as a proof by induction on the height of the derivation of the typing judgement $H, x : r, y : s, H' \vdash M : t$ since the structure of that derivation follows the structure of the term. Or, indeed, it may be viewed as an induction simply on the number of symbols appearing in M, since each subterm will have fewer symbols than the term of which it is a part. Here is another example of a proof by structural induction. The lemma will be needed to prove a crucial property of the equational theory of the lambda-calculus.

2.6 Lemma. *If $H, x : s, H' \vdash M : t$ and $H, H' \vdash N : s$, then $H, H' \vdash [N/x]M : t$.*

Proof: The proof is by structural induction on M. If $M \equiv x$, then $s \equiv t$ and the result is immediate. If M is a variable y different from x, then the desired conclusion follows from [Proj]. If M is an application, then the conclusion of the lemma follows from the inductive hypothesis and the definition of the substitution operation. Now, if $M \equiv \lambda y : u.\ L$ and $t \equiv u \to v$, then $H, x : s, H', y : u \vdash L : v$ so $H, H', y : u \vdash [N/x]L : v$ by the inductive hypothesis. Thus $H, H' \vdash [N/x]M : t$ by [Abs] and the definition of substitution. □

Equational rules.

An *equation* in the simply-typed lambda-calculus is a four-tuple (H, M, N, t) where H is a type assignment, M, N are λ-terms, and t is a type. To make a tuple like this more readable, it is helpful to replace the commas separating the components of the tuple by more suggestive symbols and to write $(H \rhd M = N : t)$. The triangular marker is intended to indicate where the interesting part of the tuple begins. The heart of the tuple is the pair of terms on either side of the of equation symbol; H and t provide typing information about these terms. An *equational theory* T is a set of equations $(H \rhd M = N : t)$ such that $H \vdash M : t$ and $H \vdash N : t$. An equation $(H \rhd M = N : t)$ should be viewed only as a formal symbol. For the judgement that an equation is *provable*, we define the relation $\vdash$ between theories T and equations $(H \rhd M = N : t)$ to be the least relation satisfying the rules in Table 2.2. The assertion $T \vdash (H \rhd M = N : t)$ is called an *equational judgement*. Of course, the turnstile symbol $\vdash$ is also used for typing judgements, but this overloading is never a problem because of the different appearance of the two forms of judgement. The two are related by the fact that if $T \vdash (H \rhd M = N : t)$, then both M and N have type t relative to H. Before proving this fact, let us look at some specific remarks about the equational judgements. For many of the rules, the best way to understand the rule is to think of the theory T, the type assignment H, and the type t as fixed. Then $\vdash$ can be viewed as defining a binary relation between terms M and N. Of course, this is the usual way to think of the equality relationship, but the indirect route taken by defining equations as tuples makes it easier to account for the role that typing judgements play in describing the semantics of terms in our subsequent discussions. An equation $(H \rhd M = N : t)$ is *provable* in theory T just in the case where $T \vdash (H \rhd M = N : t)$. A demonstration of an equational rule is called an *equation derivation*. As with typing derivation, such demonstrations have the structure of a tree labeled with equational judgements. Here are specific comments about the rules.

- The axiom rule {Axiom} asserts that all of the equations in T are provable from T.
- The addition rule {Add} asserts that the type assignment H in an equation can be extended by assignment of a type to a new variable. The drop rule {Drop} asserts that an assignment of a type to a variable can be removed from the type assignment component of an equation if the variable does not have a free occurrence in either of the term components of the equation. By the permutation rule {Permute}, any equation E obtained from an equation E' by permuting the order of variable/type pairs in the assignment component of E' is provable if E' is.
- The reflexivity rule {Refl}, symmetry rule {Sym}, and transitivity rule {Trans} assert that $=$ can be viewed as an equivalence relation on terms relative to a fixed theory and type tags.
- The congruence rule {Cong} and the ξ-rule assert that application and abstraction respectively are congruences relative to the equality relation (relative to a fixed theory and type tags).

Table 2.2
Equational Rules for the Simply-Typed λ-Calculus

{Axiom}	$\dfrac{(H \rhd M = N : t) \in T}{T \vdash (H \rhd M = N : t)}$
{Add}	$\dfrac{T \vdash (H \rhd M = N : t) \qquad x \notin H}{T \vdash (H,\ x : s \rhd M = N : t)}$
{Drop}	$\dfrac{T \vdash (H,\ x : s \rhd M = N : t) \qquad x \notin \mathrm{Fv}(M) \cup \mathrm{Fv}(N)}{T \vdash (H \rhd M = N : t)}$
{Permute}	$\dfrac{T \vdash (H,\ x : r,\ y : s, H' \rhd M = N : t)}{T \vdash (H,\ y : s,\ x : r,\ H' \rhd M = N : t)}$
{Refl}	$\dfrac{H \vdash M : t}{T \vdash (H \rhd M = M : t)}$
{Sym}	$\dfrac{T \vdash (H \rhd M = N : t)}{T \vdash (H \rhd N = M : t)}$
{Trans}	$\dfrac{T \vdash (H \rhd L = M : t) \qquad T \vdash (H \rhd M = N : t)}{T \vdash (H \rhd L = N : t)}$
{Cong}	$\dfrac{T \vdash (H \rhd M = M' : s \to t) \qquad T \vdash (H \rhd N = N' : s)}{T \vdash (H \rhd M(N) = M'(N') : t)}$
{ξ}	$\dfrac{T \vdash (H,\ x : s \rhd M = N : t)}{T \vdash (H \rhd \lambda x : s.\ M = \lambda x : s.\ N : s \to t)}$
{β}	$\dfrac{H, x : s \vdash M : t \qquad H \vdash N : s}{T \vdash (H \rhd (\lambda x : s.\ M)(N) = [N/x]M : t)}$
{η}	$\dfrac{H \vdash M : s \to t \qquad x \notin \mathrm{Fv}(M)}{T \vdash (H \rhd \lambda x : s.\ M(x) = M : s \to t)}$

- The β- and η-rules are the key to the λ-calculus. The β-rule says that the application of an abstraction to an operand is equal to the result of substituting the operand for the bound variable of the abstraction in the body of the abstraction. It is harder to explain the η-rule; it says that if M is a function, then the function that takes an argument x and applies M to it is indistinguishable from M.
- In the η-rule, the variable x cannot have a free occurrence in the term M. It is important to note the parsing in this axiom. For example, if x is not free in M, then $(\lambda x : s.\ M)x = M$ because of the β-rule. But $\lambda x : s.\ (M(x)) = M$ does *not* follow from β.

Properties of theories and equations are often proved by induction on the structure of terms, but there is another form of argument that is very common for such properties: induction on the height of a derivation tree. In general, the *height* of a tree is the length of the longest branch in the tree. To prove a property of a collection of trees by induction on the height involves checking the result for base cases—trees of height 1—and establishing that a tree of height $n + 1$ has the property if any tree of height n does. Derivations of equations in the simply-typed λ-calculus are trees where each node corresponds to an instance of a rule of the equational system given in Table 2.2. A proof by induction on the heights of such derivations can be carried out by verifying that each rule preserves the desired property in the sense that the property is satisfied by the conclusion whenever it is satisfied by its hypotheses. Here is a typical example of a proof by induction on the heights of derivation trees.

2.7 Lemma. *If T is a theory and $T \vdash (H \rhd M = N : t)$, then $H \vdash M : t$ and $H \vdash N : t$.*

Proof: This is proved by induction on the height of the derivation of the judgement $T \vdash (H \rhd M = N : t)$. The idea is to assume that the lemma holds for any equation that appears in the hypothesis of a rule from which this judgement is derived and then prove that it holds for $T \vdash (H \rhd M = N : t)$ as well. Suppose that the rule employed in the last step of the derivation is an instance of {Axiom}. Then $(H \rhd M = N : t) \in T$, and by the definition of a theory it follows that $H \vdash M : t$ and $H \vdash N : t$. Suppose that the last step of the derivation is an instance of {Permute}. In this case, Lemma 2.5 states the desired conclusion. Similar results for {Add} and {Drop} have been left as an exercise. If the last step in the proof is an instance of {Refl}, {Trans}, or {Sym}, then the desired conclusion follows immediately from the inductive hypothesis. If the last step is an instance of {Cong}, then $M \equiv M'(M'')$ and $N \equiv N'(N'')$ where, by the inductive hypothesis,

$$\begin{array}{ll} H \vdash M' : s \to t & H \vdash M'' : s \\ H \vdash N' : s \to t & H \vdash N'' : s \end{array}$$

so $H \vdash M : t$ and $H \vdash N : t$ by [Appl]. Similarly, the case of the ξ-rule follows from [Abs] and the inductive hypothesis. If the η-rule is the last step of the derivation, then $t \equiv u \to v$ for some u, v, and we must show that $H \vdash \lambda x : u.\ M(x) : v$ given that $H \vdash M : u \to v$. This follows easily if we can show that $H, x : u \vdash M : u \to v$. The proof of this is left as an exercise. Finally, if the last step of the derivation is an instance of the β-rule, then N has the form $[N'/x]M'$ where $H, x : s \vdash M' : t$ and $H \vdash N' : s$. The conclusion for N therefore follows from Lemma 2.6. For M in this case, the desired conclusion follows from [Abs] and [Appl]. □

To prove some basic facts about equations below it is essential to know more about how typing and equational judgements are related. The reader can prove as an exercise this lemma:

2.8 Lemma. *If $H \vdash M : t$ and $H' \vdash M : t$, then $H(x) = H'(x)$ for every $x \in \mathrm{Fv}(M)$.* □

This is needed to prove the following:

2.9 Lemma. *Suppose $H \vdash M : t$ and $H \vdash N : t$. If $T \vdash (H' \rhd M = N : t)$ for any type assignment H', then also $T \vdash (H \rhd M = N : t)$.*

Proof: Let G be the sublist of H' obtained by removing from H' all of the assignments of types to variables that are not in $\mathrm{Fv}(M) \cup \mathrm{Fv}(N)$. By repeated application of the {Permut} and {Drop} rules, it is possible to conclude that $T \vdash (G \rhd M = N : t)$. By Lemma 2.7, $G \vdash M : t$ and $G \vdash N : t$ so, by Lemma 2.8, $G(x) = H(x)$ for each $x \in \mathrm{Fv}(M) \cup \mathrm{Fv}(N)$. By Lemma 2.2, it must be the case that every variable in $\mathrm{Fv}(M) \cup \mathrm{Fv}(N)$ also appears in H. Since every variable in G is in $\mathrm{Fv}(M) \cup \mathrm{Fv}(N)$, it must be the case that G is a permutation of a sublist of H. Let G' be the sublist of H all of whose variables are not in $\mathrm{Fv}(M) cup \mathrm{Fv}(N)$. Then by repeated applications of {Add}, it is possible to conclude that $T \vdash (G, G' \rhd M = N : t)$. Since G, G' is just a permutation of H, it follows from repeated applications of {Permut} that $T \vdash (H \rhd M = N : t)$. □

Exercises.

2.3 Structural induction is used to prove properties of term trees as well as properties of terms.

a. Prove that $\equiv$ is an equivalence relation (Lemma 2.1).
b. Prove that substitution is well-defined on α-equivalence classes. That is, if M, M' and N, N' are term trees such that $M \equiv M'$ and $N \equiv N'$, then $[M/x]N \equiv [N'/x]M'$.

2.4 Prove Lemma 2.3.

2.5 Prove that the term $\lambda x : \mathbf{o}.\ x(x)$ does not have a type.

2.6 Complete the proof of Lemma 2.7 by proving results that apply to the rules {Add} and {Drop}:

a. Show that if $H \vdash M : t$ and $x \notin H$, then $H, x : t \vdash M : t$.
b. Show that if $H, x : s \vdash M : t$ and x is not free in M, then $H \vdash M : t$.

2.7 Prove Lemma 2.8.

2.8 For a type assignment $H = x_1 : t_1, \ldots, x_n : t_n$ and a variable x_i' not occurring in H, let $[x_i'/x_i]H$ be the assignment $x_1 : t_1, \ldots, x_i' : t_i, \ldots, x_n : t_n$. Show that if $H \vdash M = N : t$, then $[x_i'/x_i]H \vdash [x_i'/x_i]M = [x_i'/x_i]N : t$. In other words, equality is preserved under a renaming of free variables.

2.9 Note that some of the equational rules for the λ-calculus in Table 2.2 involve typing judgements $H \vdash M : t$ as well as equational judgements $T \vdash (H \rhd M = N : t)$. It is possible to modify the equational rules for the lambda-calculus so that typing judgements are not needed in the equational rules. To do this, replace the three equational rules in which typing judgements appear with the following rules:

$$\{\text{Refl}'\} \qquad \frac{x \in H}{T \vdash (H \rhd x = x : H(x))}$$

$$\{\beta'\} \qquad \frac{T \vdash (H,\ x : s \rhd M = M : t) \qquad T \vdash (H \rhd N = N : s)}{T \vdash (H \rhd (\lambda x : s.\ M)(N) = [N/x]M : t}$$

$$\{\eta'\} \qquad \frac{T \vdash (H \rhd M = M : s \to t) \qquad x \notin H}{T \vdash (H \rhd \lambda x : s.\ M(x) = M : s \to t)}$$

Prove a version of Lemma 2.7 for this new system and conclude that it is equivalent to the official set of rules in Table 2.2. Show that this new formulation makes it possible to give typing judgements as a defined notion rather than providing a separate axiomatization as in Table 2.1.

2.3 Models

There are two ways to view the relationship between an object language calculus and a model. One holds that the model is a means for studying the calculus itself. This is a common way to see things in the semantics of programming languages, since attention is very often focused on syntax. However, under another view, the calculus is a syntax for expressing elements of a distinguished model M. We then think of M as *standard* because the calculus is being judged in terms of the model. This dual view generalizes to a situation in which there is a *class* of models for a calculus where there may or may not be a distinguished standard model. This section discusses these perspectives for the semantics of the simply-typed λ-calculus. We begin with a discussion of what one might view as the standard model, in which types are interpreted as sets of functions, and then look at the definition of a class of models for the λ-calculus. It can be shown that the equational rules of the simply-typed calculus are *sound* for reasoning about the equational properties of the class of models in the sense that each equation is satisfied by all of the models in the class. Moreover, the equational rules are *complete* in the sense that an equation that is true in all of the models can actually be proved from the equational theory. At the end of the section it is shown that something more is true: an equation is provable just in the case where it holds in the standard model.

Sets and functions.

Our starting point for the discussion of models of typed calculi is the 'standard' model of the simply-typed λ-calculus in which sets are types and functions are denoted by terms. Before giving the semantic interpretation, let us pause to state the definition of a function precisely and introduce a notation for the set of functions.

Definition: A *function* (or *map*) is a relation $f : X \to Y$ such that for each $x \in X$, there is a unique $y \in Y$ such $(x, y) \in f$. For each $x \in X$, the notation $f(x)$ is used for this unique element of Y and we say that f *maps* x to y. The set of all $y \in Y$ such that $f(x) = y$ for some $x \in X$ is called the *image* of f. Given X and Y, the set of all functions with domain X and codomain Y is written Y^X. □

Now, the semantic interpretation of types is relative to a given set X, which serves as the interpretation for the base type. To keep a notational barrier between the types and terms of the λ-calculus on the one hand, and the mathematics that surrounds the semantic description on the other, syntax is ordinarily enclosed between *semantic brackets:* $[\![\cdot]\!]$. The meaning $[\![t]\!]$ of a type t is a set defined inductively as follows:

- $[\![\mathbf{o}]\!] = X$
- $[\![s \to t]\!] = [\![t]\!]^{[\![s]\!]}$.

So, for example, $[\![\mathbf{o} \to (\mathbf{o} \to \mathbf{o})]\!]$ is the set of functions f such that, for each $x \in X$, $f(x)$ is a function from X into X. On the other hand, $[\![(\mathbf{o} \to \mathbf{o}) \to \mathbf{o}]\!]$ is the set of functions F such that, for each function f from X to X, $F(f)$ is an element of X.

Describing the meanings of terms is more difficult than describing the meanings of types, and we require some further vocabulary and notation. While a type assignment associates *types* with variables, an *environment* associates *values* to variables. Environments are classified by type assignments: if H is a type assignment, then an H-*environment* is a function ρ on variables that maps each $x \in H$ to a value $\rho(x) \in [\![H(x)]\!]$. If ρ is an H-environment, $x : t \in H$, and $d \in [\![t]\!]$, then we define

$$\rho[x \mapsto d](y) = \begin{cases} d & \text{if } y \equiv x \\ \rho(y) & \text{otherwise.} \end{cases}$$

This is the 'update' operation. One can read $\rho[x \mapsto d]$ as 'the environment ρ with the value of x updated to d'. The notation is very similar to that used for syntactic substitution, but note that this operation on environments is written as a postfix. So another way to read $\rho[x \mapsto d]$ is 'the environment ρ with d for x.' Note that if $x \notin H$ for an assignment H, then $\rho[x \mapsto d]$ is an $H, x : t$ assignment if $d \notin [\![t]\!]$. Now, the meaning of a term M is described relative to a type assignment H and a type t such that $H \vdash M : t$. We use the notation $[\![H \rhd M : t]\!]$ for the meaning of term M relative to H, t. Here, as in the case of equations earlier, the triangle is intended as a kind of marker or separator between the type assignment H and the term M. We might have written $[\![H \vdash M : t]\!]$ for the meaning, but this confuses the use of $\vdash$ as a relation for typing judgements with its syntactic use as a punctuation in the expression within the semantic brackets. Nevertheless, it is important to remember that $[\![H \rhd M : t]\!]$ only makes sense if $H \vdash M : t$.

The meaning $[\![H \rhd M : t]\!]$ is a function from H-environments to $[\![t]\!]$. The semantics is defined by induction on the typing derivation of $H \vdash M : t$,

- Projection: $[\![H \rhd x : t]\!]\rho = \rho(x)$.
- Abstraction: $[\![H \rhd \lambda x : u.\ M' : u \to v]\!]\rho$ is the function from $[\![u]\!]$ to $[\![v]\!]$ given by $d \mapsto [\![H, x : u \rhd M' : v]\!](\rho[x \mapsto d])$, that is, the function f defined by

$$f(d) = [\![H, x : u \rhd M' : v]\!](\rho[x \mapsto d]).$$

- Application: $[\![H \rhd L(N) : t]\!]\rho$ is the value obtained by applying the function $[\![H \rhd L : s \to t]\!]\rho$ to argument $[\![H \rhd N : s]\!]\rho$ where s is the unique type such that $H \vdash L : s \to t$ and $H \vdash N : s$.

It will save us quite a bit of ink to drop the parentheses that appear as part of expressions such as $[\![H, x : u \rhd M' : v]\!](\rho[x \mapsto d])$ and simply write $[\![H, x : u \rhd M' : v]\!]\rho[x \mapsto d]$. Doing so appears to violate our convention of associating applications to the left, but there is little chance of confusion in the case of expressions such as these. Hence, we will adopt the convention that the postfix update operator binds more tightly than general application.

It must be shown that this assignment of meanings respects our equational rules. This is the *soundness* property of the semantic interpretation:

2.10 Theorem (Soundness). *If* $\vdash (H \rhd M = N : t)$, *then* $[\![H \rhd M : t]\!] = [\![H \rhd N : t]\!]$. □

To prove this theorem, we must check that each of the rules of the λ-calculus is satisfied. The proof proceeds by induction on the height of the derivation of an equality. The rules {Refl}, {Sym}, {Trans}, and {Cong} are immediate from the corresponding properties for equality of sets. Suppose that a proof ends by an application of the ξ-rule:

$$\frac{\vdash (H, x : s \rhd M = M' : t)}{\vdash (H \rhd \lambda x : s.\ M = \lambda x : s.\ M' : s \to t)}.$$

We apply the definition of our semantic function and the inductive hypothesis to calculate

$$\begin{aligned}
&([\![H \rhd \lambda x : s.\ M : s \to t]\!]\rho)(d) \\
&\quad = [\![H, x : s \rhd M : t]\!]\rho[x \mapsto d] \\
&\quad = [\![H, x : s \rhd M' : t]\!]\rho[x \mapsto d] \qquad (2.1) \\
&\quad = ([\![H \rhd \lambda x : s.\ M' : s \to t]\!]\rho)(d)
\end{aligned}$$

where Equation 2.1 follows from the inductive hypothesis on equational derivations. This shows that

$$[\![H \rhd \lambda x : s.\ M : s \to t]\!] = [\![H \rhd \lambda x : s.\ M' : s \to t]\!]$$

since a function is determined by its action on its arguments—a property known as *extensionality*.

To prove soundness of the η-rule we use the following:

2.11 Lemma. *Suppose* M *is a term and* $H \vdash M : t$. *If* $x \notin H$ *and* $d \in [\![s]\!]$, *then* $[\![H, x : s \rhd M : t]\!]\rho[x \mapsto d] = [\![H \rhd M : t]\!]\rho$. □

The proof is left as an exercise. The lemma essentially asserts that the meaning of a term M in a type environment H depends only on the values H assigns to free variables of M. We may therefore calculate

$$\begin{aligned}
&[\![H \rhd \lambda x : s.\ M(x) : s \to t]\!]\rho \\
&\quad = (d \mapsto [\![H, x : s \rhd M(x) : t]\!]\rho[x \mapsto d]) \\
&\quad = (d \mapsto ([\![H, x : s \rhd M : s \to t]\!]\rho[x \mapsto d])(d)) \\
&\quad = (d \mapsto ([\![H \rhd M : s \to t]\!]\rho)(d)) \\
&\quad = [\![H \rhd M : s \to t]\!]\rho
\end{aligned}$$

where the final equation follows from Lemma 2.11.

To prove soundness of the β-rule, we could begin the calculation as follows:

$$\begin{array}{rl} \multicolumn{2}{l}{[\![H \rhd (\lambda x : s.\ M)(N) : t]\!]\rho} \\ = & ([\![H \rhd \lambda x : s.\ M : s \to t]\!]\rho)([\![H \rhd N : s]\!]\rho) \\ = & [\![H, x : s \rhd M : t]\!]\rho[x \mapsto ([\![H \rhd N : s]\!]\rho)] \end{array}$$

but we now wish to conclude that this last expression is equal to $[\![H \rhd [N/x]M : t]\!]\rho$. This relationship between substitution and the 'updating' of the environment is the essence of the meaning of the β-rule. We summarize it as the following:

2.12 Lemma (Substitution). *If $H \vdash N : s$ and $H, x : s \vdash M : t$, then*

$$[\![H \rhd [N/x]M : t]\!]\rho = [\![H, x : s \rhd M : t]\!]\rho[x \mapsto ([\![H \rhd N : s]\!]\rho)].$$

Proof: Let $e = [\![H \rhd N : s]\!]\rho$. The proof of the lemma is by induction on the structure of M.

Case $M \equiv y$ where $y \not\equiv x$. Then

$$\begin{aligned} [\![H \rhd [N/x]y : t]\!]\rho &= [\![H \rhd y : t]\!]\rho \\ &= \rho(y) \\ &= [\![H, x : s \rhd y : t]\!]\rho[x \mapsto e] \end{aligned}$$

Case $M \equiv x$. In this case, $s \equiv t$. We have

$$\begin{aligned} [\![H \rhd [N/x]x : t]\!]\rho &= [\![H \rhd N : t]\!]\rho \\ &= \rho[x \mapsto e](x) \\ &= [\![H, x : s \rhd M : t]\!]\rho[x \mapsto e] \end{aligned}$$

Case $M \equiv \lambda y : u.\ M'$. We have $t \equiv u \to v$. (By the Bound Variable Convention we assume that $y \not\equiv x$ and $y \notin \mathrm{Fv}(N)$.)

$$\begin{array}{rl} \multicolumn{2}{l}{[\![H \rhd [N/x]\lambda y : u.\ M' : u \to v]\!]\rho} \\ = & [\![H \rhd \lambda y : u.\ [N/x]M' : u \to v]\!]\rho \\ = & (d \mapsto [\![H,\ y : u \rhd [N/x]M' : u \to v]\!]\rho[y \mapsto d]) \\ = & (d \mapsto [\![H,\ y : u,\ x : s \rhd M' : u \to v]\!](\rho[y \mapsto d][x \mapsto e])) \\ = & (d \mapsto [\![H,\ x : s,\ y : u \rhd M' : u \to v]\!](\rho[x \mapsto e][y \mapsto d])) \\ = & [\![H,\ x : s \rhd \lambda y : u.\ M' : u \to v]\!](\rho[x \mapsto e]) \end{array}$$

Case $M \equiv M_0(M_1)$. We have

$$[\![H \rhd [N/x](M_0(M_1)) : t]\!]\rho$$

$$\begin{aligned}
&= ([\![H \rhd [N/x]M_0 : u \to t]\!]\rho)([\![H \rhd [N/x]M_1 : u]\!]\rho) \\
&= ([\![H,\ x : s \rhd M_0 : u \to t]\!]\rho[x \mapsto e])([\![H,\ x : s \rhd M_1 : u]\!]\rho[x \mapsto e]) \\
&= [\![H,\ x : s \rhd M_0(M_1) : t]\!]\rho[x \mapsto e] \qquad \square
\end{aligned}$$

As an application of the soundness of our interpretation, consider the following:

2.13 Theorem. *The simply-typed λ-calculus is non-trivial. That is, for any type t and pair of distinct variables x and y, it is not the case that* $\vdash (x : t,\ y : t \rhd x = y : t)$.

Proof: Suppose, on the contrary, that $\vdash (x : t,\ y : t \rhd x = y : t)$. Let X be any set with more than one element and consider the model of the simply-typed λ-calculus generated by X. It is not hard to see that $[\![t]\!]$ has at least two distinct elements p and q. Now, let ρ be an $x : t,\ y : t$ environment such that $\rho(x) = p$ and $\rho(y) = q$. Then $[\![x : t,\ y : t \rhd x : t]\!]\rho = \rho(x) = p \neq q = \rho(y) = [\![x : t,\ y : t \rhd y : t]\!]\rho$. But this contradicts the soundness of our interpretation. $\square$

It is instructive, as an exercise on the purpose of providing a semantic interpretation for a calculus, to try proving Theorem 2.13 directly from first principles and the rules for the λ-calculus using syntactic means. The soundness result provides us with a simple way of demonstrating properties of the rules of our calculus or, dually, a syntax for proving properties of our model (sets and functions).

Type frames.

Although we have given a way to associate a 'meaning' $[\![H \rhd M : t]\!]$ to a tuple, and demonstrated that our assignment of meaning preserves the required equations, we did not actually provide a rigorous description of the ground rules for saying when such an assignment really is a *model* of the simply-typed λ-calculus. In fact, there is more than one way to do this, depending on what one considers important about the model. The choice of definition may be a matter of style or convenience, but different choices may also reflect significant distinctions. Indeed, many different definitions were presented and studied in the late 1970's, (especially for the *untyped* λ-calculus, which we will discuss later). The definition provided in this section is, perhaps, the simplest and most intuitive one, although not always the easiest to check and not the most general. An 'environment model' is an interpretation that assigns a meaning to a term M with respect to an environment ρ. The model we gave above is an example of a presentation in this form. The notion of an environment is designed to deal with the meaning of an expression that has free variables. Although every model of the λ-calculus must deal with the meanings of free variables, there are other ways to handle them that reflect a different style of description. In particular, another definition of model, based on the idea of a *category,* will be given later.

For the sake of convenience, our first definition of a model of the simply-typed λ-calculus is broken into two parts. Models are called frames; these are defined in terms of a more general structure called a pre-frame.

Definition: A *pre-frame* is a pair of functions $\mathcal{A}[\![\cdot]\!]$ and A on types and pairs of types respectively such that

- $\mathcal{A}[\![t]\!]$ is a non-empty set, which we view as the interpretation of type t, and
- $A^{s,t} : \mathcal{A}[\![s \to t]\!] \times \mathcal{A}[\![s]\!] \to \mathcal{A}[\![t]\!]$ is a function that we view as the interpretation of the application of an element of $\mathcal{A}[\![s \to t]\!]$ to an element of $\mathcal{A}[\![s]\!]$,

and such that the *extensionality property* holds: that is, whenever $f, g \in \mathcal{A}[\![s \to t]\!]$ and $A^{s,t}(f, x) = A^{s,t}(g, x)$ for every $x \in \mathcal{A}[\![s]\!]$, then $f = g$. □

To make the notation less cumbersome, we write $(\mathcal{A}, A)$ for a pre-frame and use $\mathcal{A}$ to represent the pair. Pre-frames are very easy to find. For example, we might take $\mathcal{A}[\![s]\!]$ to be the set of natural numbers for every s and define $A^{s,t}(f, x)$ to be the product of f and x. Since $f * 1 = g * 1$ implies $f = g$, the extensionality property is clearly satisfied. Nevertheless, this multiplication pre-frame does not provide any evident interpretation for λ-terms (indeed, there is none satisfying the equational rules—see Exercise 2.12).

A frame is a pre-frame together with a sensible interpretation for λ-terms.

Definition: A *type frame* (or *frame*) is a pre-frame $(\mathcal{A}^{\text{type}}, A)$ together with a function $\mathcal{A}^{\text{term}}$ defined on triples $H \rhd M : t$ such that $H \vdash M : t$. An *H-environment* is a function ρ from variables to meanings such that $\rho(x) \in \mathcal{A}^{\text{type}}[\![H(x)]\!]$ whenever $x \in H$. $\mathcal{A}^{\text{term}}[\![H \rhd M : t]\!]$ is a function from H-environments into $\mathcal{A}^{\text{type}}[\![t]\!]$. The function $\mathcal{A}^{\text{term}}[\![\cdot]\!]$ is required to satisfy the following equations:

1. $\mathcal{A}^{\text{term}}[\![H \rhd x : t]\!]\rho = \rho(x)$
2. $\mathcal{A}^{\text{term}}[\![H \rhd M(N) : t]\!]\rho = A^{s,t}(\mathcal{A}^{\text{term}}[\![H \rhd M : s \to t]\!]\rho,\ \mathcal{A}^{\text{term}}[\![H \rhd N : s]\!]\rho)$
3. $A^{s,t}(\mathcal{A}^{\text{term}}[\![H \rhd \lambda x : s.\ M : s \to t]\!]\rho,\ d) = \mathcal{A}^{\text{term}}[\![H,\ x : s \rhd M : t]\!]\rho[x \mapsto d]$. □

If a pre-frame has an extension to a frame, then the extension is unique.

2.14 Lemma. *Let $(\mathcal{A}^{\text{type}}, A)$ be a pre-frame over which $\mathcal{A}^{\text{term}}[\![\cdot]\!]$ and $\bar{\mathcal{A}}^{\text{term}}[\![\cdot]\!]$ define frames. Then $\mathcal{A}^{\text{term}}[\![H \rhd M : t]\!] = \bar{\mathcal{A}}^{\text{term}}[\![H \rhd M : t]\!]$ whenever $H \vdash M : t$.*

Proof: The proof is by induction on the structure of M. The only non-trivial case occurs when $t \equiv r \to s$ and $M \equiv \lambda x : r.\ N$. In this case,

$$\begin{aligned} A^{r,s}(\mathcal{A}^{\text{term}}[\![H \rhd M : r \to s]\!]\rho,\ d) &= \mathcal{A}^{\text{term}}[\![H,\ x : r \rhd N : s]\!]\rho[x \mapsto d] \\ &= \bar{\mathcal{A}}^{\text{term}}[\![H,\ x : r \rhd N : s]\!]\rho[x \mapsto d] \\ &= A^{r,s}(\bar{\mathcal{A}}^{\text{term}}[\![H \rhd M : r \to s]\!]\rho,\ d) \end{aligned}$$

so the desired result follows from the extensionality property for pre-frames. □

In the future I will use the same notation $\mathcal{A}$ for both $\mathcal{A}^{\text{type}}[\![\cdot]\!]$ and $\mathcal{A}^{\text{term}}[\![\cdot]\!]$. The lemma says that the former together with an application operation A determines the latter, so it simplifies matters to write a pair $(\mathcal{A}, A)$ for a frame. When the application operation A is not explicitly mentioned or when it is understood from context, I may simply write $\mathcal{A}$ for the frame itself.

A frame $\mathcal{A}$ should be viewed as a model of the λ-calculus; we write

$$\mathcal{A} \models (H \rhd M = N : t)$$

if, and only if, $\mathcal{A}[\![H \rhd M : t]\!]\rho = \mathcal{A}[\![H \rhd N : t]\!]\rho$ for each H-environment ρ. Whenever it will not cause confusion, it helps to drop the type tags and type and write $\mathcal{A} \models M = N$. If T is a set of equations, then

$$\mathcal{A} \models T$$

if, and only if, $\mathcal{A} \models (H \rhd M = N : t)$ for each equation $(H \rhd M = N : t)$ in T. Define $T \models M = N$ if $\mathcal{A} \models M = N$ whenever $\mathcal{A} \models T$.

The most basic example of a frame is the interpretation we discussed in the previous section. Given a set X, the *full frame over* X is $\mathcal{F}_X = (\mathcal{F}_X[\![\cdot]\!], F_X)$ where

- $\mathcal{F}_X[\![\mathbf{o}]\!] = X$ and $\mathcal{F}_X[\![s \to t]\!]$ is the set of functions from $\mathcal{F}_X[\![s]\!]$ to $\mathcal{F}_X[\![t]\!]$
- $F_X^{s,t}(f, x) = f(x)$, that is, $F_X^{s,t}$ is ordinary function application,
- on terms, $\mathcal{F}_X[\![\cdot]\!]$ is the function $[\![\cdot]\!]$ defined in the previous section.

It is easy to see that our definition of the semantic function $[\![\cdot]\!]$ corresponds exactly to the three conditions in the definition of a frame. Moreover, these were essentially the properties that made our proof of the soundness property for the interpretation possible. To be precise:

2.15 Theorem (Soundness for Frames). *For any theory T and frame $\mathcal{A}$, if $\mathcal{A} \models T$ and $T \vdash (H \rhd M = N : t)$, then $\mathcal{A} \models (H \rhd M = N : t)$.* □

The proof of the Theorem is left as an exercise; it is very similar to the proof of the soundness of the full type frame. When T is empty, we have the following:

2.16 Corollary. *For any frame $\mathcal{A}$, if $\vdash M = N$, then $\mathcal{A} \models M = N$* $\square$

Example: A binary relation $\sqsubseteq$ on a set P is said *anti-symmetric* if $x = y$ whenever $x \sqsubseteq y$ and $y \sqsubseteq x$. A *partial order* or *poset* is a set P together with a binary relation $\sqsubseteq$ that is reflexive, transitive, and anti-symmetric. If $(P, \sqsubseteq_P)$ and $(Q, \sqsubseteq_Q)$ are partial orders, a function $f : P \to Q$ is *monotone* if $x \sqsubseteq_P y$ implies $f(x) \sqsubseteq_Q f(y)$. Given posets P and Q, the monotone functions $f : P \to Q$ between P and Q are themselves a poset; the *pointwise ordering* on these functions is defined by taking $f \sqsubseteq g$ iff $f(x) \sqsubseteq_Q g(x)$ for each $x \in P$. Given a poset P, there is a frame $\mathcal{A}$ where $\mathcal{A}[\![\mathbf{o}]\!] = P$ and $\mathcal{A}[\![s \to t]\!]$ is the poset of monotone functions from $\mathcal{A}[\![s]\!]$ to $\mathcal{A}[\![t]\!]$ under the pointwise ordering. Application for this frame is the usual function application. $\square$

Another important class of examples of frames can be formed from equivalence classes of well-typed terms of the simply-typed calculus. To define these frames we need some more notation for type assignments. An *extended* type assignment $\mathcal{H} = x_1 : t_1,\ x_2 : t_2, \ldots$ is an infinite list of pairs such that every finite subset $H \subseteq \mathcal{H}$ is a type assignment and every type appears infinitely often. Note that if $H \vdash M : t$ and $H' \vdash M : s$ where $H, H' \subseteq \mathcal{H}$, then $s \equiv t$. Now, fix an extended type assignment $\mathcal{H}$. Let T be a set of equations of the form $(H \rhd M = N : t)$ where $H \subseteq \mathcal{H}$. If $H \vdash M : t$ for some $H \subseteq \mathcal{H}$, define

$$[M]_T = \{M' \mid T \vdash (H' \rhd M = M' : t) \text{ for some } H' \subseteq \mathcal{H}\}.$$

This defines an equivalence relation on such terms M (the proof is left as an exercise). When T is the empty set, we drop the subscript T. For each type t, define

$$\mathcal{T}_T[\![t]\!] = \{[M]_T \mid H \vdash M : t \text{ for some } H \subseteq \mathcal{H}\}.$$

For each pair of types s, t, define $\mathrm{TermAppl}_T^{s,t} : \mathcal{T}_T[\![s \to t]\!] \times \mathcal{T}_T[\![s]\!] \to \mathcal{T}_T[\![t]\!]$ by

$$\mathrm{TermAppl}_T^{s,t}([M]_T, [N]_T) = [M(N)]_T.$$

This is well-defined because of the congruence rule for application. These operations define what is called the *term* pre-frame:

2.17 Lemma. *The pair $(\mathcal{T}_T, \mathrm{TermAppl}_T)$ is a pre-frame.*

Proof: Suppose $[f]_T, [g]_T \in \mathcal{T}_T[\![s \to t]\!]$ and $[f(M)]_T = [g(M)]_T$ for all M of type s. Let $x : s \in \mathcal{H}$ be a variable that has no free occurrence in f or g. Then $[f(x)]_T = [g(x)]_T$, so $T \vdash (H \rhd f(x) = g(x) : t)$ for some $H \subseteq \mathcal{H}$. By ξ-rule (together with the rules for permuting and adding, to be precise), we have

$$T \vdash (H \rhd \lambda x : s.\ f(x) = \lambda x : s.\ g(x) : s \to t).$$

Hence, by the η-rule, $T \vdash (H \rhd f = g : s \to t)$. Thus $[f]_T = [g]_T$. $\square$

We now show that this pre-frame is a frame, which is called the *term model* over T. To do this we begin by extending our earlier definition of a substitution to permit *simultaneous* substitutions. We write $\sigma = [M_1, \ldots, M_n/x_1, \ldots, x_n]$ for the function that maps the variable x_i to the term M_i for each i and acts as the identity on other variables. It is assumed that $x_1, \ldots, x_n$ are distinct. The *support* of the substitution is the set of variables on which the substitution is not the identity; of course, the support of $[M_1, \ldots, M_n/x_1, \ldots, x_n]$ is a subset of $\{x_1, \ldots, x_n\}$. The substitution $\sigma = [M_1, \ldots, M_n/x_1, \ldots, x_n]$ can be extended to substitution on terms by inductively defining

- $\sigma(M(N)) \equiv (\sigma(M))(\sigma(N))$
- $\sigma(\lambda x : t.\ M) \equiv \lambda x : t.\ \sigma(M)$ where x is not in the support of σ or in $\mathrm{Fv}(\sigma(y))$ for any y in the support of σ.

This generalizes our earlier notation $[M/x]$ which may now be viewed as a substitution with support $\{x\}$. When x is not in the support of σ, we write $\sigma[x \mapsto M]$ or $\sigma[M/x]$ for $[M_1, \ldots, M_n, M/x_1, \ldots, x_n, x]$.

Let ρ be an H-environment for the term pre-frame: that is, $\rho(x) \in \mathcal{T}_T[\![H(x)]\!]$ whenever $x \in H$. Let us say that a substitution σ *represents* ρ *over* H if, for each x in H, the term $\sigma(x)$ is a representative of the term model equivalence class $\rho(x)$.

2.18 Lemma. *Let* $\mathcal{T}_T[\![H \rhd M : t]\!]\rho = [\sigma(M)]_T$ *where* σ *is a substitution representing* ρ *over* H*. Then* $(\mathcal{T}_T, \mathrm{TermAppl}_T)$ *is a type frame.*

Proof: That $\mathcal{T}_T[\![\cdot]\!]$ is well-defined follows from the congruence of application and abstraction with respect to equality. To prove that we have a frame, the appropriate conditions for variables, applications, and abstractions must be established. I will leave the variable and application cases to the reader and prove the necessary condition for abstractions.

Suppose σ represents ρ over H, then

$$\begin{array}{rl}
& \mathrm{TermAppl}_T^{s,t}(\mathcal{T}_T[\![H \rhd \lambda x : s.\ M : s \to t]\!]\rho,\ [N]_T) \\
= & \mathrm{TermAppl}_T^{s,t}([\sigma(\lambda x : s.\ M : s \to t)]_T,\ [N]_T) \\
= & \mathrm{TermAppl}_T^{s,t}([\lambda x : s.\ \sigma(M) : s \to t]_T,\ [N]_T) \\
= & [(\lambda x : s.\ \sigma(M))(N)]_T \\
= & [(\sigma[N/x])M]_T \\
= & \mathcal{T}_T[\![H,\ x : s \rhd M : t]\!]\rho[x \mapsto [N]_T]
\end{array}$$

since $\sigma[N/x]$ represents $\rho[x \mapsto [N]_T]$ over $H,\ x : s$. □

We are now ready to see the proof of the main result of this section:

2.19 Theorem (Completeness for Frames). *$T \vdash M = N$ if, and only if, $T \models M = N$.*

This follows immediately from Lemma 2.18 and the following:

2.20 Theorem. *$T \vdash (H \rhd M = N : t)$ if, and only if, $\mathcal{T}_T \models (H \rhd M = N : t)$.*

Proof: Necessity ($\Rightarrow$) follows immediately from the Soundness Theorem 2.15 for frames and the fact that the term model is a frame (Lemma 2.18).

To prove sufficiency ($\Leftarrow$), rename the variables in H so that $H \subseteq \mathcal{H}$ (see Exercise 2.8). Choose ρ to be the identity environment $\rho : x \mapsto [x]_T$. The identity substitution $\sigma : x \mapsto x$ represents this over H. Now, $[M]_T = [\sigma(M)]_T = \mathcal{T}_T[\![H \rhd M : t]\!]\rho = \mathcal{T}_T[\![H \rhd N : t]\!]\rho = [\sigma(N)] = [N]_T$ so $T \vdash (H' \rhd M = N : t)$ for some $H' \subseteq \mathcal{H}$. Hence, by Lemma 2.9, $T \vdash (H \rhd M = N : t)$ as well. □

A particularly important example of a frame in the class of term models is the one induced by the empty theory: $\mathcal{T}_\emptyset$. For this particular term model it is convenient to drop the subscript $\emptyset$. As an instance of Theorem 2.20, we have the following:

2.21 Corollary. *$\vdash M = M'$ if, and only if, $\mathcal{T} \models M = M'$.* □

Completeness of the full type frame.

Given a collection of mathematical structures, it is usually fruitful to find and study collections of structure-preserving transformations or mappings between them. Homomorphisms of algebras are one such example, and continuous maps on the real numbers another example. What kinds of mappings between type frames should we take to be 'structure-preserving'? The definition we seek for the goal of this section is obtained by following the spirit of homomorphisms between algebras but permitting *partial* structure-preserving mappings and requiring such maps to be surjective. This will provide the concept needed to prove that the full type frame is complete.

Definition: A *partial function* $f : A \rightsquigarrow B$ is a subset of $A \times B$ such that whenever $(x, y), (x, z) \in f$, then $y = z$. The *domain of definition* of f is the set of $x \in A$, such that $(x, y) \in f$ for some $y \in B$. For a partial function f, we write $f(x) = y$ if, and only if, $(x, y) \in f$. A partial function $f : A \rightsquigarrow B$ is a *surjection* if, for any $y \in B$, there is some $x \in A$ such that $(x, y) \in f$. $\Box$

Notation: Another common notation used to indicate that a set $f \subseteq A \times B$ is a partial function is to write $f : A \rightharpoonup B$ using a kind of half arrow. This is easier to write than $\rightsquigarrow$ but has the disadvantage of looking too much like $\rightarrow$. A simple notation to write that is almost as pleasing to read as the official one given in the definition above is to put a tilde on top of an arrow to emphasize the possibility of partiality; one writes $f : A \overset{\sim}{\rightarrow} B$. $\Box$

Definition: Let $\mathcal{A}$ and $\mathcal{B}$ be frames. A *partial homomorphism* $\Phi : \mathcal{A} \rightarrow \mathcal{B}$ is a family of surjective partial functions $\Phi^s : \mathcal{A}[\![s]\!] \rightsquigarrow \mathcal{B}[\![s]\!]$ such that, for each s, t, and $f \in \mathcal{A}[\![s \rightarrow t]\!]$ either

1. there is some $g \in \mathcal{B}[\![s \rightarrow t]\!]$ such that
$$\Phi^t(A^{s,t}(f, x)) = B^{s,t}(g, \Phi^s(x)) \tag{2.2}$$
for all x in the domain of definition Φ^s and $\Phi^{s \rightarrow t}(f) = g$, or
2. there is no element $g \in \mathcal{B}[\![s \rightarrow t]\!]$ that satisfies Equation 2.2 and $\Phi^{s \rightarrow t}(f)$ is undefined. $\Box$

Suppose that g and h are solutions to Equation 2.2. Then $B^{s,t}(g, y) = B^{s,t}(h, y)$ for each $y \in \mathcal{B}[\![s]\!]$ since Φ^s is a surjection. Extensionality therefore implies that g and h are equal. So, if there is a solution in $\mathcal{B}[\![s \rightarrow t]\!]$ for Equation 2.2, then there is a unique one.

The following is the basic fact about partial homomorphisms:

2.22 Lemma. *Let $\mathcal{A}$ and $\mathcal{B}$ be frames. If $\Phi : \mathcal{A} \rightarrow \mathcal{B}$ is a partial homomorphism and ρ is an H-environment for $\mathcal{A}$ and ρ' is an H-environment for $\mathcal{B}$ such that $\Phi^t(\rho(x)) = \rho'(x)$ for each variable x in H, then*
$$\Phi^t(\mathcal{A}[\![H \triangleright M : t]\!]\rho) = \mathcal{B}[\![H \triangleright M : t]\!]\rho'$$
whenever $H \vdash M : t$.

Proof: The proof proceeds by induction on the structure of M.

Case $M \equiv x$. Then $\Phi^t(\mathcal{A}[\![H \triangleright x : t]\!]\rho) = \Phi^t(\rho(x)) = \rho'(x) = \mathcal{B}[\![H \triangleright x : t]\!]\rho'$.

Case $M \equiv L(N)$.

$$\begin{aligned}
&\Phi^t(\mathcal{A}[\![H \rhd L(N) : t]\!]\rho) \\
&\quad = \Phi^t(A^{s,t}(\mathcal{A}[\![H \rhd L : s \to t]\!]\rho,\ \mathcal{A}[\![H \rhd N : s]\!]\rho)) \\
&\quad = B^{s,t}(\Phi^{s\to t}(\mathcal{A}[\![H \rhd L : s \to t]\!]\rho),\ \Phi^t(\mathcal{A}[\![H \rhd N : s]\!]\rho)) \\
&\quad = B^{s,t}(\mathcal{B}[\![H \rhd L : s \to t]\!]\rho',\ \mathcal{B}[\![H \rhd N : s]\!]\rho') \\
&\quad = \mathcal{B}[\![H \rhd L(N) : t]\!]\rho'.
\end{aligned}$$

Case $M \equiv \lambda x : u.\ M'$ where $t \equiv u \to v$. If d is in the domain of Φ^u, then

$$\begin{aligned}
&B^{u,v}(\mathcal{B}[\![H \rhd \lambda x : u.\ M' : u \to v]\!]\rho', \Phi^u(d)) \\
&\quad = \mathcal{B}[\![H,\ x : u \rhd M' : v]\!]\rho'[x \mapsto \Phi^u(d)] \\
&\quad = \Phi^v(\mathcal{A}[\![H,\ x : u \rhd M' : v]\!]\rho[x \mapsto d]) \\
&\quad = \Phi^v(A^{u,v}(\mathcal{A}[\![H \rhd \lambda x : u.\ M' : u \to v]\!]\rho,\ d)).
\end{aligned}$$

The conclusion of the lemma therefore follows directly from the condition on $\Phi^{u\to v}$ in the definition of a partial homomorphism.

2.23 Lemma. *If there is a partial homomorphism $\Phi : \mathcal{A} \to \mathcal{B}$, then*

$$\mathcal{A} \models (H \rhd M = N : t) \textit{ implies } \mathcal{B} \models (H \rhd M = N : t).$$

Proof: Suppose ρ' is an H-environment for $\mathcal{B}$. Choose ρ so that $\rho'(x) = \Phi^t(\rho(x))$ for each x in H. This is possible because Φ^s is a surjection. Then

$$\begin{aligned}
\mathcal{B}[\![H \rhd M : t]\!]\rho' &= \Phi^t(\mathcal{A}[\![H \rhd M : t]\!]\rho) \\
&= \Phi^t(\mathcal{A}[\![H \rhd N : t]\!]\rho) \\
&= \mathcal{B}[\![H \rhd N : t]\!]\rho'.
\end{aligned}$$
□

2.24 Lemma. *Let $\mathcal{A}$ be a type frame and suppose there is a surjection from a set X onto $\mathcal{A}[\![\mathbf{o}]\!]$. Then there is a partial homomorphism from $\mathcal{F}_X$ (the full type frame over X) to $\mathcal{A}$.*

Proof: Let $\Phi^{\mathrm{o}} : X \to \mathcal{A}[\![\mathbf{o}]\!]$ be any surjection. Suppose $\Phi^s : \mathcal{F}_X[\![s]\!] \to \mathcal{A}[\![s]\!]$ and $\Phi^t : \mathcal{F}_X[\![t]\!] \to \mathcal{A}[\![t]\!]$ are partial surjections. We define $\Phi^{s\to t}(f)$ to be the unique element of $\mathcal{A}[\![s \to t]\!]$, if it exists, such that $A^{s,t}(\Phi^{s\to t}(f), \Phi^s(y)) = \Phi^t(f(y))$ for all y in the domain of definition of Φ^s. Proof that this defines a surjection is carried out by induction on structure of types. It holds by assumption for ground types; suppose $g \in \mathcal{A}[\![s \to t]\!]$ and Φ^s, Φ^t are surjections. Choose $g' \in \mathcal{F}_X[\![s \to t]\!] = \mathcal{F}_X[\![s]\!] \to \mathcal{F}_X[\![t]\!]$ such that, for all y in the domain of definition of Φ^s, we have $g'(y) \in (\Phi^t)^{-1}(A^{s,t}(g, \Phi^s(y)))$. This is possible because Φ^t is a surjection. Since $\mathcal{A}$ is a type frame, extensionality implies that $\Phi^{s\to t}(g') = g$. By the definition of Φ^s it is therefore a partial homomorphism. □

2.25 Theorem (Completeness for Full Type Frame). *If X is infinite, then*

$$\vdash (H \rhd M = N : t)$$

if, and only if,

$$\mathcal{F}_X \models (H \rhd M = N : t).$$

Proof: We proved soundness ($\Rightarrow$) earlier. To prove sufficiency ($\Leftarrow$), begin by noting that Lemma 2.24 implies that there is a a partial homomorphism from the full type frame, $\mathcal{F}_X$, onto the term model, $\mathcal{T}$. If $\mathcal{F}_X \models (H \rhd M = N : t)$, then $\mathcal{T} \models (H \rhd M = N : t)$ by Lemma 2.23. By Theorem 2.20, this means that $\vdash (H \rhd M = N : t)$, the desired conclusion. □

Exercises.

2.10 Prove Lemma 2.11.

2.11 Let $t \equiv \mathbf{o} \to \mathbf{o}$. Show it is not the case that

$$\vdash \lambda f : t.\ \lambda x : \mathbf{o}.\ f(f(x)) = \lambda f : t.\ \lambda x : \mathbf{o}.\ f(x).$$

2.12 Prove that the 'multiplication' pre-frame, which takes $\mathcal{A}[\![s]\!]$ to be the set of natural numbers for every s and $A^{s,t}(m, n) = m * n$, is not a type frame.

2.13 Let $\mathcal{H}$ be an extended assignment and let T be a theory. Given terms M and N, say $M \sim N$ if there is some $H \subseteq \mathcal{H}$ and type t such that $T \vdash (H \rhd M = M' : t)$. Prove that $\sim$ is an equivalence relation on such terms. Note that this equivalence relation determines the equivalence classes $[M]_T$ in the term model.

2.14 Prove the Soundness Theorem for frames, Theorem 2.15.

2.15* Show that Theorem 2.25 fails when X is not infinite.

Definition: Let $\mathcal{A}$ and $\mathcal{B}$ be frames. A family of relations R^s indexed over types s is said to be a *logical relation* between $\mathcal{A}$ and $\mathcal{B}$ if

- $R^s \subseteq \mathcal{A}[\![s]\!] \times \mathcal{B}[\![s]\!]$.
- $f\ R^{s\to t}\ g$ if, and only if, the following condition holds for every x, y: if $x\ R^s\ y$, then $A^{s,t}(f, x)\ R^t\ B^{s,t}(g, y)$. □

2.16 The notion of a partial homomorphism is a special instance of a logical relation. Explain why this is the case and conjecture a version of Lemma 2.22 about partial homomorphisms that would apply to logical relations. Prove your version of the lemma.

2.4 Notes

A basic introduction to the simply-typed λ-calculus can be found in the book of Hindley and Seldin [114]. Much of the material in the last section of this chapter draws on the paper of Friedman [84] in which the Completeness Theorem for the full type frame is proven. The notion of a partial homomorphism comes from that paper, but the generalization of partial homomorphisms called a *logical relation* defined in Exercise 2.16 is a fundamental tool in the study of the simply-typed λ-calculus. Many of the basic properties of logical relations were developed by Tait, Statman, and Howard [118; 244; 245; 246; 247; 255], and they continue to be a topic of interest for applications. A general survey on logical relations is included in a handbook article of Mitchell [175]. A paper by Burn, Hankin, and Abramsky [41] furnishes an example of how logical relations can be applied to the static analysis of programs.