The Semantics of Types in Programming Languages

Carl A. Gunter

Contents

1 Introduction

In the twentieth century, there have been at least two lines of development of the notion of a *type.* One of these uses types to conquer problems in the foundations of mathematics. For example, type distinctions can resolve troubling paradoxes that lead to inconsistent systems. But a second, more recent, line of investigation into types pursues their application in programming languages. Although computer architectures themselves suggest few type distinctions, 'higher-level' programming languages generally use a classification of data into types to serve a variety of different purposes. There are at least four motives for doing this.

One of the earliest reasons for using types in programming languages such as Fortran was the enhancement of efficiency. For example, if a variable is declared to be an array of integers having a given number of entries, then it is possible to provide good space management for the variable in computer memory. A programmer's indication of the type of a datum might also save pointless testing in a running program. This use of types has remained a basic motivation for their presence in many modern languages and will remain an important application.

A second motivation for the use of types was appreciated by the end of the 1960's. This was their role as a programming *discipline.* Types could be used to enforce restrictions on the shape of well-formed programs. One key benefit in doing this was the possibility of detecting flaws in programs in the form of compiletime type errors. If a mistake in a program can be detected at the point that the program is submitted for compilation into machine code, then this mistake can be corrected promptly rather than being discovered later in a run of the program on data. Although the software engineering gains obtained in this way are widely recognized, a price is also paid for them. First, by constraining the programmer with a type system, some apparently reasonable programs will be rejected even though their runtime behavior would be acceptable. Second, the typing system may demand extensive programmer annotations, which can be time-consuming to write and tedious to read. Reducing the impact of these two drawbacks is the central objective of much of the work on types in programming languages.

A third motivation for types in programming languages is the one most recently understood: their role in supporting data abstraction and modularity. Although these cornerstones of software engineering principle can be achieved to some extent without types, many programming languages employ a type system that enforces the hiding of data type representations and supports the specification of modules. For example, several languages support a separation between a package specification, which consists of a collection of type declarations, and the body of the package, which provides programs implementing the procedures, *etc.* that appear in the specification. These units are automatically analyzed to determine their type-correctness before code is generated by compilation.

A fourth motivation for the use of types, and the primary topic of this chapter, is their role as a conceptual tool for classifying programs in a way that permits a more abstract understanding of their meanings. For virtually any language, a semantics will classify objects according to their structure and use, thereby elevating them conceptually above the sequences of bits that they will be compiled into. This abstraction is the most fundamental use of type systems.

2 Types in Programming

This section discusses a collection of programming examples intended to illustrate the motivations for various type structures. A handy pair of languages for making a comparison is Scheme, which can be viewed as based on the *untyped* λ-calculus, and ML, which can be viewed as based on the *typed* λ-calculus. Both languages have specifications that are at least as clear as one finds for most languages, so it is (usually) not difficult to tell what the meaning of a program in the language is actually specified to be. (An IEEE standard for Scheme was introduced in 1990 [IEE, 1991]; for ML the *Standard* version [Milner *et al.*, 1990] is used in the examples below.) While both were designed with semantic clarity as a key objective, the designs are also sensitive to efficiency issues, and both have good compilers that are widely used. Finally, the functional fragments of Scheme and ML employ a call-by-value evaluation strategy so the chance of confusing operational differences with differences in the type system philosophy are reduced.

2.1 Higher types.

One of the first discoveries of researchers investigating the mathematical semantics of programming languages in the late 1960's was the usefulness of *higher-order* functions in describing the denotations of programs. The use of higher-order functions in *programming* was understood even earlier and incorporated in the constructs and programming methodology of Lisp. Programmers using the languages Scheme and ML employ higher-order functions as a tool for writing clear, succinct code by capturing good abstractions. On the other hand, the use of higher-order functions does have its costs, and it is therefore worthwhile to discuss some of the ways in which such functions can be useful. Their usefulness as a tool in the semantics of programming languages is adequately argued by books and articles that employ them extensively in semantic descriptions ([Tennent, 1992] provides a starting point.) Rather than review the subject from that perspective, let us instead consider briefly why they are useful in programming.

Consider a familiar mathematical operation: taking the derivative of a continuous real-valued function. The derivative of a function f is the function f' where

$$f'(x) = \frac{f(x + dx) - f(x)}{dx}$$

for an infinitesimal dx. For purposes of an estimate, it will simplify our discussion to bind dx to a small number (say .0001). Of course, it could passed as a parameter, but this is not necessary for the point below. A Scheme program for computing the derivative can be coded as follows:

```
(define (deriv f x)
  (/ (- (f (+ x dx)) (f x))
     dx))
```

Here the derivative is higher-order, since it takes a function as a parameter, but it is coded as returning a numerical value on a given argument, rather than returning a new function as is the case for the mathematical derivative. This can lead to problems if the distinction is not properly observed. For example, the program `(deriv (deriv f 1))`, which might be mistakenly intended to compute the second derivative of `f` at 1, will yield a *runtime type error* complaining that `deriv` has the wrong number of arguments. Of course, the second derivative at 1 could be successfully calculated using an explicit abstraction,

```
(deriv (lambda (x) (deriv f x)) 1).
```

and the 'lambda' could be eliminated by making a local definition if this intrusion of lambda-abstraction is considered undesirable. However, these approaches generalize poorly to the case where what is wanted is the third derivative or the fourth derivative, and so on. To accommodate these cases, it would be possible to include a parameter `n` for the n'th iteration of differentiation in the definition of `deriv`, but it is more elegant and understandable to quit fighting against the mathematical usage in the programming, and to start coding it instead. The derivative takes a function as an argument and produces a function as a value:

```
(define (deriv f)
  (lambda (x)
    (/ (- (f (+ x dx)) (f x))
       dx)))
```

The second derivative at 1 is now properly coded as

```
((deriv (deriv f)) 1)
```

as in the mathematical notation $f''(1)$ where the primes denote an operation on a function. To calculate n'th derivatives, it is possible to write another function that takes a function g and a number n as arguments and produces the function $g^n = g \circ g \circ \cdots \circ g$ (n copies) as its value. This is a powerful abstraction since there are many other ways this function might be used; the key idea here, the composition exponential, can be used modularly with the derivative rather than being mixed up in the code for the n'th derivative function.

Where do types come into this? Lying at the heart of the distinction just discussed is the notion of 'currying', that is, the passage from a function of type $r \times s \to t$ to one of type $r \to (s \to t)$. In the first case above, the derivative function was coded as a function taking as its arguments a real-valued function `f` and a real number `x`. When coded in ML, it looks like this:

```
fun deriv (f:real -> real, x:real):real
  = (f(x+dx) - f(x))/dx.
```

and the ML type-checker indicates its type as

```
((real -> real) * real) -> real
```

The way to read this is to think of it as the definition of a function `deriv` on a product type with the abstraction described using pattern matching. The second (curried) way to program this function is

```
fun deriv f = fn x:real => ((f(x+dx) - f(x))/dx):real
```

where the ML syntax for λ is `fn`. For this term, the type is

```
(real -> real) -> (real -> real)
```

Another example of the usefulness of higher-order functions comes from the powerful programming techniques one can obtain by combining them with references and assignments. An example drawn from [Abelson and Sussman, 1985] appears in Table 1. The procedure `make-account` takes a starting balance as an argument and produces an 'account object' as a value. The account object is itself a higher-order function, returned as `dispatch`, that has its own local state given by the contents of the 'instance variable' `balance`, which contains the current balance of the object. The arguments taken by the object include the 'messages' represented by atoms `'withdraw` and `'deposit` and the arguments of the 'message sends', which appear in the formal parameters `amount` in the 'method definitions' of `withdraw` and `deposit`. To create an account, the balance of the object

Table 1. Using Local Variables and Higher-Order Functions in Scheme

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (sequence (set! balance
                        (- balance amount))
                  balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request"
                m))))
 dispatch)
```

must be initialized by applying `make-account` to the number that will be the starting value of its instance variable. For example, if `Dan` is defined as the value of `(make-account 50)` and `George` is defined as the value of `(make-account 100)`, then the account objects will correctly maintain their separate balance levels though a sequence of 'messages sends' describing the financial history of the objects. Finding a suitable system of types to classify programs written in 'object-oriented' style is a major area of research as this chapter is being written.

2.2 Recursive types.

Consider the following pair of programs:

```
(define (cbvY f)
  ((lambda (x) (lambda (y) (f (x x) y)))
   (lambda (x) (lambda (y) (f (x x) y)))))

(define (badadd x) (+ x "astring")).
```

Both of them are compiled without comment by Scheme. They illustrate a trade-off in the type-checking of programs. The first program `cbvY` is the *call-by-value fixed-point combinator.* It is an interesting program that can be used to make recursive definitions without using explicit recursive equations. The second program `badadd` contains a 'semantic' error in the form of a bad addition. Many of the actual bugs encountered in programs are like this one—despite how silly it looks in this simple example. Both of these programs can be executed in Scheme although the second program will probably cause a runtime type error. These two examples could be rendered in ML as follows:

```
fun cbvY f =
  ((fn x => (fn y => f (x x) y))
   (fn x => (fn y => f (x x) y)))

fun badadd x = x + "astring".
```

but the ML type-checker will reject both of them as having type errors. In the second case the rejection will occur because there is an expression that purports to add something to a string—an operation that is not allowed. The first program is rejected because it has an application `x x` of a variable to itself and no type is be inferred for this by the ML type-checker. From the viewpoint of a programmer, one might see the type-checking as a useful diagnostic for the second program and a hindrance for the first one.

A programming language is said to have *static* type-checking if the type-correctness of the program is established before the program is compiled. Scheme does not carry out such a check, but ML does. Of course, type-correctness is relative to the typing discipline, so this is a language design issue. In a language that allows most or all programs to be accepted as type-correct at compiletime, it will be necessary to carry out various runtime type checks, and some errors that might have been caught by a type-checker may not be detected as quickly as one would desire. At an alternate extreme, a way to ensure that there are no runtime type errors is to reject all programs as having type errors at compiletime. Of course, no programming language has a typing discipline as strict as that, but many languages are more restrictive than seems reasonable. The right balance in a language will discipline programming in order to provide useful diagnostic testing for errors while not ruling out programs that capture useful abstractions or efficient algorithms.

For example, languages that check the types of programs before compiling them can compensate for the problem with `cbvY` just mentioned by employing a more general type inference system than that of ML or by allowing the programmer to use explicit *recursive types*. The ML version of the call-by-value fixed-point combinator is given in Table 2. To understand this program, think of it as an annotation of the earlier program with coercions that make the types of subterms explicit. In fact, the type of the ML program is

Table 2. ML Version of the Call-by-Value Fixed-Point Combinator

```
local
  datatype 'a fix = FUN of 'a fix -> 'a
  fun NUF (FUN x) = x
in
  fun cbvY f =
    (fn x => (fn y => f((NUF x) x)y))
    (FUN(fn x => (fn y => f((NUF x) x)y)))
end
```

$$((a \to b) \to (a \to b)) \to (a \to b)$$

rather than $(c \to c) \to c$ (where a, b, c are type variables) as one might have hoped, but rather than carry out a detailed analysis of the program, let us just consider a couple of points about it. First of all, the definition of the procedure `cbvY` itself in the three lines of code between `in` and `end` is *not* recursive since `cbvY` appears only on the left-hand side of the defining equation. The recursion lies instead in the `datatype` declaration in the local bindings where a unary operator `fix` (written in postfix notation) is defined by a recursive equation. In that expression, the symbol `'a` represents a *type variable.* A more mathematical way of writing the datatype declaration would be to indicate that $\mathit{fix}(a) \cong \mathit{fix}(a) \to a$ where

$$FUN : (\mathit{fix}(a) \to a) \to \mathit{fix}(a)$$

defines the isomorphism. The inverse of the isomorphism is the function

$$NUF : \mathit{fix}(a) \to (\mathit{fix}(a) \to a)$$

defined in the third line of the program. If we remove the declaration of the type and the isomorphisms from this ML program, we obtain the definition of `cbvY` that was rejected as having type errors.

2.3 Parametric polymorphism.

Although recursive types are a powerful tool for recovering the losses incurred by imposing a type discipline, there is another subtle concept to be found in the way certain abstractions can be formed in the untyped language. Here is a Scheme program that appends two lists of elements:

```
(define (append headlist taillist)
  (if (null? headlist)
      taillist
      (cons (car headlist)
            (append (cdr headlist) taillist))))
```

Scheme programmers never need to concern themselves about the *types* of the elements in the lists being appended, since this program will work equally well on any pair of arguments so long as they are *lists.* In some languages where programmers must declare their types, it is impossible to obtain this level of abstraction. Instead, it may be necessary to write a program that appends lists of integers and another program that appends lists of string arrays, and so on.

To avoid losing abstractions, languages with static type-checking deal with this problem by using *polymorphism.* The word 'polymorphism' means having many forms; in typed programming languages it ordinarily refers to the idea that a symbol may have many types. In one of its simplest forms, polymorphism arises from the use of a variable that specifies an indeterminate or parameterized type for an expression. This is called *parametric polymorphism.* A basic form of parametric polymorphism views this as a kind of macro expansion. For example, the Ada programming language has a construct known as a *generic* that serves this purpose. A procedure declared with a generic in Ada is explicitly instantiated with a type before it is used, but the abstraction can make it unnecessary to rewrite a piece of code that would work equally well for two different types. For example, the function that appends lists takes a pair of lists of elements of type t as an argument and returns a list of elements of type t as a result. Here the particular type t is unimportant, so it is replaced by a variable a and the type is indicated as $\mathrm{list}(a) \to \mathrm{list}(a)$. To see another example, consider the function that takes two reference cells and exchanges their contents. Obviously, this operation is independent of the types of elements in the reference cells; it is a 'polymorphic swapping' function. In ML it can be coded as follows:

```
fun swap (x,y) =
  let val temp = ! x
  in  x := ! y ; y := temp
  end
```

where the exclamation marks are the dereferencing operation: `!x` denotes the contents of the reference cell `x`. A novelty of the ML programming language is an inference algorithm that can *infer* a polymorphic type for programs without the need for programmer annotations. Specifically, using ML syntax, the type is inferred to be

```
swap : ('a ref * 'a ref) -> unit.
```

The function `swap` works with a side effect (change of memory); its output is unimportant so it is taken to be the unique value of type `unit`. The type of `swap` indicates that the references have the same type since the type variable `'a` is used for *both* arguments. This means that it is type-correct to swap the contents of two integer references or swap the contents of two string references, but a program that might swap the contents of an integer reference with that of a string reference will be rejected with a type error before being compiled.

Let us anticipate the precise definition of ML polymorphism with some discussion of what its limitations are in programming. Although type inference is an excellent tool for cutting the tedium of providing type annotations for programs, there is a great deal of abstraction that is lost in the compromises of the ML polymorphic types. Consider, for example, the following Scheme program:

```
(define applyto
  (lambda (f) (cons (f 3) (f "hi"))))
```

It defines a procedure `applyto` which takes a function as an argument and forms a cons cell from the results of applying it to the number `3` and the string `"hi"`. While it should not be difficult to think up many interesting things that can be applied to both `3` and `"hi"`, let us keep things simple and consider

```
(applyto (lambda (x) x))
```

which evaluates to the cell `(3 . "hi")`. All this seems very simple and natural, but the ML type inference algorithm is unwilling to see this program as type-correct. In particular, the program

```
fn f => ( f(3), f("hi") )
```

will be rejected with an indication that the function `f` cannot take both `3` *and* `"hi"` as arguments since they have different types. This seems a bit dull-witted in light of the Scheme example, which evidently shows that there are perfectly good programs that *can* take both of these as arguments. ML has a construct that allows some level of such polymorphism. The program

```
let fun I x = x in (I(3), I("hi")) end
```

is type-correct but clearly fails to achieve the abstraction of the Scheme program since it only makes sense for a *given* value of `f` (in this case, `f` is `I`). To obtain a program as abstract as the one written in Scheme, it is necessary to introduce a more expressive type system than the one ML has. The Girard-Reynolds polymorphic λ-calculus, which is presented in a later section, has the desired expressiveness.

2.4 Subtypes.

Another kind of programming language polymorphism that is being used in many modern languages is based on the notion of a *subtype.* This is a form of type polymorphism that arises from the classification of data according to collections of *attributes.* This perspective draws its inspiration from hierarchical systems of categories such as the taxonomy of the animal kingdom rather than from the variation of a parameter as in quantifiers of predicate logic.

To get some of the spirit of this kind of typing, let us begin with an informal example based on the kind of hierarchy that one might form in order to classify some of the individuals one finds at a university; let us call such individuals *academics.* Each academic has an associated *university* and *department* within the university. At the university there are *professors,* who teach courses, and *students,* who attend the courses taught by the professors. Some of the students are *employees* of the university in a capacity as *teaching assistants (TA's)* while others are *research assistants (RA's)* supported by research grants. Each of these various classes of individuals has associated attributes. For instance, if we consider a typical semester, we can attribute to professors and teaching assistants the courses they are teaching—their *teaching load.* In this capacity as teachers, the professors and TA's are employees and therefore have a *salary* associated with them. RA's have a *project* attribute for the research project on which they are working.

To bring some order to this assortment of groups and attributes, it is helpful to organize a hierarchy of groups *classified by their defining attributes.* Let us begin to list each group and its attributes. First of all, we could use a type of *persons,* whose members, which include both academics and employees, have a *name* attribute. In addition to a name, each academic has a university and each employee has a *salary,* a *social security number* (for tax purposes), and an *employer.* In addition to attributes inherited from their roles as academics, each student has an *advisor* and each professor has a teaching load and a boolean *tenure* attribute. Professors are also employees, so they must possess the attributes of employees as well as those of academics. We can now classify our assortment by using common attributes to form the poset based on the relations

$$\begin{array}{l}\mathrm{Academic}, \mathrm{Employee} \leq \mathrm{Person}\\ \mathrm{Student}, \mathrm{Professor} \leq \mathrm{Academic}\\ \mathrm{RA}, \mathrm{TA}, \mathrm{Professor} \leq \mathrm{Employee}\\ \mathrm{RA}, \mathrm{TA} \leq \mathrm{Student}\end{array}$$

together with those relations obtained from an assumption that $\leq$ is transitive and reflexive. Each point in the poset represents a *type* of individual based on attributes the individual must possess. If a type t is greater than a type s in the poset, this means that each kind of attribute that an individual of type of t possesses must also be had by each individual of type s. If $s \leq t$, we say that s is a *subtype* of t. The fact that a professor must have a social security number is something one can conclude by the fact that the type of professors is a subtype of that of employees and the fact that each employee has a social security number.

Each of the types in the example given above can be viewed as a kind of product where the components of a tuple having that type are its attributes. In programming languages these are generally called *records,* and the attributes are called the *fields* of the record. Records are usually written with curly brackets '{' and '}' rather than with parentheses as tuples are. Semantically they are very similar to tuples, but the field labels relieve the need to write the record fields in any particular order. A common record syntax is a sequence of pairs of the form $l = M$ where l is a *label* and M is the term associated with that label. The term M is generally called the *l-field* of the record. For example, the records

```
{Name = "Carl Gunter",
 University = "University of Pennsylvania"}

{University = "University of Pennsylvania",
 Name = "Carl Gunter"}
```

are considered equivalent, and the type of these records is given by the following equivalent pair of record type expressions:

```
{Name : String, University : String}
{University : String, Name : String}.
```

Now, we would like to mix records such as these with the dual notion of a *variant.* They are written with square (as opposed to curly) brackets '[' and ']'. For instance, a biological classification system might include a declaration such as

```
type ReproductiveSystem = [Male : MaleSystem,
                           Female : FemaleSystem]
```

defining a familiar partition of the collection of reproductive systems. In this expression, `Male` and `Female` are labels for the fields of the variant; the types of these fields must be `MaleSystem` and `FemaleSystem` respectively. The order in which the fields are written is insignificant. A classification system for vehicles might have a type

```
type Vehicle = [Air : AirVehicle,
```

Table 3. Declarations for a Subtype Hierarchy

```
type Thing = {Age : Int}
type Machine = Thing + {Fuel : String}
type MovingMachine = Machine + {MaxSpeed : Int}
type AirVehicle =
  MovingMachine +
  {MaxAltitude : Int, MaxPassengers : Int}
type LandVehicle =
  MovingMachine +
  {Surface : String, MaxPassengers : Int}
type WaterVehicle =
  MovingMachine +
  {Tonnage : Int, MaxPassengers : Int}
```

```
                Land : LandVehicle,
                Water : WaterVehicle]
```

in which vehicles are classified according to their preferred milieu. A term of this type would come from one of the three possible components. For example,

```
[Air = SouthernCross]
```

is a term of type `Vehicle` if `SouthernCross` is a term of type `AirVehicle`. And

```
[Water = QueenMary]
```

is also a term of type `Vehicle` if `QueenMary` is a term of type `WaterVehicle`. To see a little more detail for these types, consider the declarations in Table 3. Here each of the defined types is a record type. To make the notation more succinct, a plus sign is written to indicate, for instance, that a `Machine` is a record having a field `Fuel` together with all of the fields had by a `Thing` (namely an `Age` field). Consider what a subtype of type `Vehicle` might be. In the case of records, a subtype has more fields than a supertype. In a variant, the dual holds. For instance,

```
type WheeledVehicle = [Air : WheeledAirVehicle,
                       Land : WheeledLandVehicle]
```

is a subtype of `Vehicle` where

```
type WheeledLandVehicle = LandVehicle +
                             {WheelsNumber : Int}
type WheeledAirVehicle = AirVehicle +
                         {WheelsNumber : Int}.
```

Intuitively, a wheeled vehicle is either an air vehicle with wheels or a land vehicle with wheels. If we forget about the wheels, then a wheeled vehicle can be viewed simply as a vehicle. This example also illustrates that it is not just the fact that there are fewer fields that matters for variants, but that the types of the fields that exist are subtypes of the corresponding fields from the supertype. Looking at this from the point of view of a term of type `WheeledVehicle`, note that

```
value MyCar = [Land = {Age = 3,
                       Fuel = "Gasoline",
                       MaxSpeed = 100,
                       Surface = "Roadway",
                       MaxPassengers = 5,
                       WheelsNumber = 4}]
```

has the type `Vehicle` if the last field, which indicates the number of wheels, is omitted.

This provides some intuition about the subtyping relation between records and between variants, but there is still one more type constructor to which we would like to generalize the idea: the function space operator. Suppose, for instance, that we need a function

```
Using : String -> Machine
```

which, given a kind of fuel (described by a string), returns an example of a `Machine` that uses that fuel. In any context where such a function is needed, we could just as easily use a function

```
WaterVehicleUsing : String -> WaterVehicle,
```

which, given a kind of fuel (described by a string), returns an example of a `WaterVehicle` that uses that fuel. This suffices because a `WaterVehicle` is a kind of machine.

Suppose now that we need a function having the type

```
HowSoon : {Start : Place, Finish : Place,
           Mode : AirVehicle} -> Int
```

where the type `Place` is a record consisting of a latitude and a longitude and the function calculates a lower bound on how soon the given mode of transport could make it from `Start` to `Finish`. Suppose we have on hand a function

```
MovingMachineHowSoon : {Start : Place, Finish : Place,
                              Mode : MovingMachine} -> Int
```

which calculates a value from its arguments in the naive way using the distance between the two places and the maximum speed of a `MovingMachine` as an argument. This can be used to serve the purpose of `HowSoon` since an `AirVehicle` is a special kind of `MovingMachine`. The method used to calculate `HowSoon` on an instance of the latter type also applies to an instance of the former.

These examples suggest that we should take `String -> Machine` to be a subtype of `String -> WaterVehicle` and take

```
{Start : Place, Finish : Place, Mode : MovingMachine}
   -> Int
```

to be a subtype of

```
{Start : Place, Finish : Place, Mode : AirVehicle}
   -> Int.
```

In the general case we will want to generalize this by taking $s \to t$ to be a subtype of $s' \to t'$ just in case t is a subtype of t' and s' is a subtype of s. Note the change in the ordering with respect to the first arguments: if $s \to t \leq s' \to t'$, then $s' \leq s$ rather than $s \leq s'$.

3 Simple Types as Sets

The simply-typed λ-calculus is the most basic of the typed calculi with higher-order functions. It is described as a collection of terms and types together with independent systems of typing judgements and equational judgements. The types t and terms M are given by the following grammar

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

where Variable is a (possibly infinite) collection of primitive syntactic objects called *variables.* In the discussions below, 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 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. Terms are generally written with letters L, M, N. Such letters annotated 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. Terms 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 modulo some standard parsing conventions. For types, the *association of the operator* $\rightarrow$ *is to the right:* for instance, $\mathbf{o} \rightarrow \mathbf{o} \rightarrow \mathbf{o}$ 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:* for instance, 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.

Terms are treated as equivalent up to the renaming of bound variables (α-equivalence). To avoid tedious repetitions of assumptions about the names of bound variables, it is helpful to use the

Convention 3.0.1 (Bound Variable Naming Convention). When a term representing an α-equivalence class is chosen, the name of the bound variable of the representative is taken to be distinct from the names of free variables in other terms being discussed.

Syntactic identity between terms is denoted by the relation $\equiv$. Given terms M and N and a variable x, the expression $[M/x]N$ is the term obtained by substituting M for x in N. This must be done modulo renaming bound variables in N to avoid capturing free variables of M.

3.1 Types and equations.

To describe the typing system for the simply-typed λ-calculus, some notation for associating types with free variables is required. 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. 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* (or *appears*) 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. This relation between H, M, and t is written in the form $H \vdash M : t$ and read 'in the assignment H, the term M has type t'. It is defined to be the least

Table 4. 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]	$\dfrac{H \vdash M : s \to t \qquad H \vdash N : s}{H \vdash M(N) : t}$

relation satisfying the axiom and two rules in Table 4. A demonstration of $H \vdash M : t$ from these rules is called a *typing derivation.* We have the following basic fact about this system:

Lemma 3.1.1. *If $H \vdash M : t$ and x does not appear in H, then x is not free in M.*

In general, we will only be interested in terms M and type assignments H such that $H \vdash M : t$ for some type t. A term M is said to be *untypeable* if there is no type assignment H and type t such that $H \vdash M : t$. For example, $\lambda x : \mathbf{o}.\ x(x)$ fails to have a type in any type assignment. If a term has a type in a given assignment, that type is unique in the following sense:

Lemma 3.1.2. *If $H \vdash M : s$ and $H \vdash M : t$, then $s \equiv t$.*

Type tags are placed on bound variables in abstractions just to make Lemma 3.1.2 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 3.1.2 would *fail.* For example, we would then have

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

as well as

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

Some further important properties of the type system are the following:

Lemma 3.1.3. *If $H,\ x : r,\ y : s,\ H' \vdash M : t$, then $H,\ y : s,\ x : r,\ H' \vdash M : t$.*

Lemma 3.1.4. *If $H, x : s, H' \vdash M : t$ and $H, H' \vdash N : s$, then $H, H' \vdash [N/x]M : t$.*

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 5. 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 following fact:

Lemma 3.1.5. *If T is a theory and $T \vdash (H \rhd M = N : t)$, then $H \vdash M : t$ and $H \vdash N : t$.*

Another basic property is the following:

Lemma 3.1.6. *Suppose $H \vdash M : t$ and $H \vdash N : t$. Let H' be a type assignment such that $H'(x) = H(x)$ for each $x \in \mathrm{Fv}(M) \cup \mathrm{Fv}(N)$. If $T \vdash (H' \rhd M = N : t)$, then also $T \vdash (H \rhd M = N : t)$.*

Further discussion of the typed λ-calculus, including many of its interesting syntactic properties can be found in [Hindley and Seldin, 1986] or in [Barendregt, 1992].

3.2 Sets as a model.

The 'standard' model of the simply-typed λ-calculus interprets types as sets where higher types are the sets of functions between sets. The semantics is relative to the choice of a set X, which serves as the interpretation for the base type. The meaning $[\![t]\!]$ of a type t is a set defined inductively as follows:

Table 5. 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}	$\dfrac{T \vdash (H \rhd M = M' : s \to t) \qquad T \vdash (H \rhd N = N' : s)}{T \vdash (H \rhd M(N) = M'(N') : t)}$
$\{\xi\}$	$\dfrac{T \vdash (H,\ x : s \rhd M = N : t)}{T \vdash (H \rhd \lambda x : s.\ M = \lambda x : s.\ N : s \to t)}$
$\{\beta\}$	$\dfrac{H, x : s \vdash M : t \qquad H \vdash N : s}{T \vdash (H \rhd (\lambda x : s.\ M)(N) = [N/x]M : t)}$
$\{\eta\}$	$\dfrac{H \vdash M : s \to t \qquad x \notin \mathrm{Fv}(M)}{T \vdash (H \rhd \lambda x : s.\ M(x) = M : s \to t)}$

- $[\![\mathbf{o}]\!] = X$
- $[\![s \to t]\!] = \{f \mid f \text{ is a function from } [\![s]\!] \text{ to } [\![t]\!]\}$.

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 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 $H, x : t$ environment if $d \in [\![t]\!]$. 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 the 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 can be shown that this assignment of meanings respects our equational rules. This is the *soundness* property of the semantic interpretation:

Theorem 3.2.1 (Soundness). *If* $\vdash (H \rhd M = N : t)$, *then* $[\![H \rhd M : t]\!] = [\![H \rhd N : t]\!]$.

This is proved by induction on the height of a derivation tree for an equational judgements by examining each case for the last rule employed. For example, the soundness of the η-rule depends on the following fact:

Lemma 3.2.2. *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 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 third equality follows from Lemma 3.2.2.

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

Theorem 3.2.3. *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. ∎

It is instructive, as an exercise on the purpose of providing a semantic interpretation for a calculus, to try proving Theorem 3.2.3 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).

3.3 Type frames.

Although we have given a way to associate a 'meaning' $[\![H \rhd M : t]\!]$ to a triple H, M, t such that $H \vdash M : t$ and demonstrated that our assignment of meaning preserves the required equations from Table 5, 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. The form of model described in this section is generally refered to as an 'extensional environment model'. The discussion follows the treatment of [Friedman, 1975] and the primary objective is to discuss the two completeness theorems that he proves there (given as Theorems 3.3.8 and 3.4.5 below).

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

Definition 3.3.1. 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, the reader may wish to try the exercise of proving that there is none satisfying the equational rules).

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

Definition 3.3.2. A *type frame* (or *frame*) is a pre-frame $(\mathcal{A}^{\mathrm{type}}, A)$ together with a function $\mathcal{A}^{\mathrm{term}}$ 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 $\rho(x) \in \mathcal{A}^{\mathrm{type}}[\![H(x)]\!]$ whenever $x \in H$. $\mathcal{A}^{\mathrm{term}}[\![H \rhd M : t]\!]$ is a function from H-environments into $\mathcal{A}^{\mathrm{type}}[\![t]\!]$. The function $\mathcal{A}^{\mathrm{term}}[\![\cdot]\!]$ is required to satisfy the following equations:

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

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

Lemma 3.3.3. *Let $(\mathcal{A}^{\mathrm{type}}, A)$ be a pre-frame over which $\mathcal{A}^{\mathrm{term}}[\![\cdot]\!]$ and $\bar{\mathcal{A}}^{\mathrm{term}}[\![\cdot]\!]$ define frames. Then $\mathcal{A}^{\mathrm{term}}[\![H \rhd M : t]\!] = \bar{\mathcal{A}}^{\mathrm{term}}[\![H \rhd M : t]\!]$ whenever $H \vdash M : t$.*

In general, it is therefore convenient to use the same notation $\mathcal{A}$ for both $\mathcal{A}^{\mathrm{type}}[\![\cdot]\!]$ and $\mathcal{A}^{\mathrm{term}}[\![\cdot]\!]$. 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.

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 typing information 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 'standard' frame uses sets and functions: 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:

Theorem 3.3.4 (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 $\mathcal{A} \models (H \rhd M = N : t)$.*

When T is empty, we have the following:

Corollary 3.3.5. *For any frame $\mathcal{A}$, if $\vdash M = N$, then $\mathcal{A} \models M = N$*

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 prefix $H \subseteq \mathcal{H}$ is a type assignment and every type appears infinitely often (that is, for each type t, there are infinitely many varaibles x such that $x : t$ appears in $\mathcal{H}$). 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 us say that a theory T is an *$\mathcal{H}$-theory* if $H \subseteq \mathcal{H}$ for each $(H \rhd M = N : t) \in T$. Let T be an $\mathcal{H}$-theory. 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. It can be shown that

Lemma 3.3.6. *The pair $(\mathcal{T}_T, \mathrm{TermAppl}_T)$ is a pre-frame.*

Indeed, this pre-frame is a frame, which is called the *term model* over T. To see this we need a notation for *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 $\rho(x)$.

Lemma 3.3.7. *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.*

Type frames form a complete class of models for theories of the simply-typed calculus:

Theorem 3.3.8 (Completeness for Frames). *$T \vdash M = N$ if, and only if, $T \models M = N$.*

This follows immediately from Lemma 3.3.7 and the following:

Theorem 3.3.9. *Suppose $H \subseteq \mathcal{H}$ and T is an $\mathcal{H}$-theory, then $T \vdash (H \rhd M = N : t)$ if, and only if, $\mathcal{T}_T \models (H \rhd M = N : t)$.*

Proof. Necessity follows immediately from the Soundness Theorem 3.3.4 for frames and the fact that the term model is a frame. To prove sufficiency, 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 3.1.6, $T \vdash (H \rhd M = N : t)$ 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 3.3.9, we have the following:

Corollary 3.3.10. *$\vdash M = M'$ if, and only if, $\mathcal{T} \models M = M'$.*

3.4 Completeness for sets.

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

1. there is some $g \in \mathcal{B}[\![s \to t]\!]$ such that
$$\Phi^t(A^{s,t}(f, x)) = B^{s,t}(g, \Phi^s(x)) \qquad (3.1)$$
for all x in the domain of definition of Φ^s and $\Phi^{s\to t}(f) = g$, or
2. there is no element $g \in \mathcal{B}[\![s \to t]\!]$ that satisfies Equation 3.1 and $\Phi^{s\to t}(f)$ is undefined.

Suppose that g and h are solutions to Equation 3.1. 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 $\mathcal{B}[\![s \to t]\!]$ for Equation 3.1, then there is a unique one.

The following is the basic fact about partial homomorphisms; it implies as a corollary the preservation of equations by partial homomorphisms.

Lemma 3.4.2. *Let $\mathcal{A}$ and $\mathcal{B}$ be frames. If $\Phi : \mathcal{A} \to \mathcal{B}$ 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*

$$\Phi^t(\mathcal{A}[\![H \rhd M : t]\!]\rho) = \mathcal{B}[\![H \rhd M : t]\!]\rho'$$

whenever $H \vdash M : t$.

Corollary 3.4.3. *If there is a partial homomorphism $\Phi : \mathcal{A} \to \mathcal{B}$ and $\mathcal{A} \models (H \rhd M = N : t)$, then $\mathcal{B} \models (H \rhd M = N : t)$.*

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 $\mathcal{B}[\![H \rhd M : t]\!]\rho' = \Phi^t(\mathcal{A}[\![H \rhd M : t]\!]\rho)$ and $\Phi^t(\mathcal{A}[\![H \rhd N : t]\!]\rho) = \mathcal{B}[\![H \rhd N : t]\!]\rho'$ by Lemma 3.4.2. But $\mathcal{A}[\![H \rhd M : t]\!]\rho$ and $\mathcal{A}[\![H \rhd N : t]\!]\rho$ are equal by assumption. ∎

Lemma 3.4.4. *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 $\Phi^{\mathbf{o}} : X \to \mathcal{A}[\![\mathbf{o}]\!]$ be any surjection. Suppose

$$\begin{array}{l}\Phi^s : \mathcal{F}_X[\![s]\!] \to \mathcal{A}[\![s]\!] \\ \Phi^t : \mathcal{F}_X[\![t]\!] \to \mathcal{A}[\![t]\!]\end{array}$$

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. ∎

Theorem 3.4.5 (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 3.4.4 implies that there is a a partial homomorphism from the full type frame, $\mathcal{F}_X$, onto the term model, $\mathcal{T}$ where $\mathcal{H}$ is chosen so that $H \subseteq \mathcal{H}$. If $\mathcal{F}_X \models (H \rhd M = N : t)$, then $\mathcal{T} \models (H \rhd M = N : t)$ by Corollary 3.4.3. By Theorem 3.3.9, this means that $\vdash (H \rhd M = N : t)$, the desired conclusion. ∎

Partial homomphisms are a special instance of a more general notion called a *logical relation* which serves as the one of the most basic tools for reasoning about types. Many of the properties of logical relations were developed by Tait, Statman, and Howard [Howard, 1973; Statman, 1982; Statman, 1985a; Statman, 1985b; Statman, 1986; Tait, 1967], and they continue to be a topic of interest for applications. A general survey on logical relations is included in [Mitchell, 1990], and [Burn *et al.*, 1986] furnishes an example of how logical relations can be applied to the static analysis of programs.

4 Simple Types as Domains

The simply-typed λ-calculus is too primitive to serve as a programming language: a programming language typically provides a notion of *evaluation* rather than simply an equational theory. Even when a directed use of equations for the simply-typed calculus is taken as an operational semantics, the resulting language is somewhat unexpressive. When evaluation is considered, the structures needed to model types must account for the semantics of such computational concepts as divergence and recursive definitions. This section considers how the theory of domains is related to the semantics of types for a basic language that extends the simply-typed calculus with a judicious collection of primitives in order to provide something like the power of a programming language.

4.1 A Programming Language for Computable Functions.

The system known as PCF (**P**rogramming language for **C**omputable **F**unctions) was introduced by Dana Scott [Scott, 1969]. The variant of Scott's system described here is taken from [Breazu-Tannen *et al.*, 1990]. Its types and terms are defined as follows:

$$\begin{array}{lll} t & ::= & \mathbf{num} \mid \mathbf{bool} \mid t \to t \\ M & ::= & \mathbf{0} \mid \mathbf{true} \mid \mathbf{false} \mid \\ & & \mathbf{succ}(M) \mid \mathbf{pred}(M) \mid \mathbf{zero?}(M) \mid \mathbf{if}\ M\ \mathbf{then}\ M\ \mathbf{else}\ M \mid \\ & & x \mid \lambda x : t.\ M \mid MM \mid \mu x : t.\ M \end{array}$$

The syntax of PCF essentially includes the terms of the simply-typed λ-calculus but with two ground types **num** and **bool**. Conventions for PCF syntax are similar to those for the basic simply-typed calculus.

Typing rules for PCF are those of the simply-typed λ-calculus (Table 4) together with those given in Table 6. Two basic facts about the type system are given by the following:

Lemma 4.1.1.

1. *If* $H \vdash M : s$ *and* $H \vdash M : t$, *then* $s \equiv t$.
2. *If* $H, x : s \vdash M : t$ *and* $H \vdash N : s$, *then* $H \vdash [N/x]M : t$.

Table 6. Typing Rules for PCF

[Zero]	$H \vdash \mathbf{0} : \mathbf{num}$
[True]	$H \vdash \mathbf{true} : \mathbf{bool}$
[False]	$H \vdash \mathbf{false} : \mathbf{bool}$
[Pred]	$\dfrac{H \vdash M : \mathbf{num}}{H \vdash \mathbf{pred}(M) : \mathbf{num}}$
[Succ]	$\dfrac{H \vdash M : \mathbf{num}}{H \vdash \mathbf{succ}(M) : \mathbf{num}}$
[IsZero]	$\dfrac{H \vdash M : \mathbf{num}}{H \vdash \mathbf{zero?}(M) : \mathbf{bool}}$
[Cond]	$\dfrac{H \vdash L : \mathbf{bool} \qquad H \vdash M : t \qquad H \vdash N : t}{H \vdash \mathbf{if}\ L\ \mathbf{then}\ M\ \mathbf{else}\ N : t}$
[Rec]	$\dfrac{H,\ x : t \vdash M : t}{H \vdash \mu x : t.\ M : t}$

Parts (1) and (2) are the analogs of Lemmas 3.1.2 and 3.1.4 respectively.

4.2 Operational semantics.

To see PCF as a programming language we need to describe how its well-typed programs are evaluated. One approach to describing such a semantics is to indicate how a term M evaluates to another term M' by defining a relation $M \rightarrow M'$ between closed terms using a set of *evaluation rules.* The goal of such rewriting is to obtain a *value* to which no further rules apply. In order to define precisely how a term is related to a value, we define a binary *transition relation* $\rightarrow$ to be the least relation on pairs of PCF terms that satisfies the axioms and rules in Table 7. A set of evaluation rules given in this form is sometimes called a *Structural Operational Semantics (SOS)* because the hypotheses of the rules involve only the evaluation of proper structural components of the expressions in their conclusions. This approach to semantics was developed by Gordon Plotkin [Plotkin, 1976; Plotkin, 1981]. We say that a term M evaluates to a value V just in the case $M \rightarrow^* V$ where $\rightarrow^*$ is the transitive, reflexive closure of the transition relation and a value is a term generated by the following grammar:

Table 7. Transition Rules for Call-by-Name Evaluation of PCF

$$\frac{M \to N}{\mathbf{pred}(M) \to \mathbf{pred}(N)} \qquad \mathbf{pred}(\mathbf{0}) \to \mathbf{0} \qquad \mathbf{pred}(\mathbf{succ}(V)) \to V$$

$$\frac{M \to N}{\mathbf{zero?}(M) \to \mathbf{zero?}(N)}$$

$$\mathbf{zero?}(\mathbf{0}) \to \mathbf{true} \qquad \mathbf{zero?}(\mathbf{succ}(V)) \to \mathbf{false}$$

$$\frac{M \to N}{\mathbf{succ}(M) \to \mathbf{succ}(N)}$$

$$\frac{M \to N}{M(L) \to N(L)} \qquad (\lambda x : t.\ M)(N) \to [N/x]M$$

$$\mathbf{if\ true\ then}\ M\ \mathbf{else}\ N \to M \qquad \mathbf{if\ false\ then}\ M\ \mathbf{else}\ N \to N$$

$$\frac{L \to L'}{\mathbf{if}\ L\ \mathbf{then}\ M\ \mathbf{else}\ N \to \mathbf{if}\ L'\ \mathbf{then}\ M\ \mathbf{else}\ N}$$

$$\mu x : t.\ M \to [\mu x : t.\ M/x]M$$

$$V \ ::= \ \mathbf{0} \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{succ}(V) \mid \lambda x : t.\ M \qquad (4.1)$$

Letters U, W range over values. The transition relation is deterministic:

Lemma 4.2.1. *If $M \to N$ and $M \to N'$ then $N \equiv N'$.*

One way to emphasize the structurality of the rules in Table 7 is to represent them using a grammar. An *evaluation context* for PCF is described by the following grammar

$$E \ ::= \ [\,] \mid \mathbf{pred}(E) \mid \mathbf{zero?}(E) \mid \mathbf{succ}(E) \mid E(L) \mid \mathbf{if}\ E\ \mathbf{then}\ M\ \mathbf{else}\ N$$

where $[\,]$ is intended to represent a 'hole' in a PCF term. A term of PCF is obtained from an evaluation context by filling the 'hole' in the context by a term: this is written in the from $E[M]$. (A examination of the grammar reveals that a context E has exactly one 'hole' in it.) If we now take analogs of the *axioms* (as opposed to the *rules*) from Table 7:

$$\begin{array}{l}
\mathbf{pred}(\mathbf{0}) \Rightarrow \mathbf{0} \\
\mathbf{pred}(\mathbf{succ}(V)) \Rightarrow V \\
\mathbf{zero?}(\mathbf{0}) \Rightarrow \mathbf{true} \\
\mathbf{zero?}(\mathbf{succ}(V)) \Rightarrow \mathbf{false} \\
(\lambda x : t.\ M)(N) \Rightarrow [N/x]M \\
\mathbf{if\ true\ then}\ M\ \mathbf{else}\ N \Rightarrow M \\
\mathbf{if\ false\ then}\ M\ \mathbf{else}\ N \Rightarrow N
\end{array}$$

then the desired relation is defined by using the following rule:

$$\frac{M \Rightarrow N}{E[M] \rightarrow E[N]}.$$

This approach to describing a structural operational semantics was introduced in [Felleisen and Friedman, 1986].

There are other approaches to describing the evaluation of a programming language. One idea is to describe the relation $M \rightarrow^* V$ more directly using a new set of rules. Such a description is sometimes known as a *natural (operational) semantics* because a semantics given in this form resembles a natural deduction system for a logic. An early instance of such a semantics appears in [Martin-Löf, 1971] and more recent examples in [Clément *et al.*, 1986; Kahn, 1987] and the Standard for ML [Milner *et al.*, 1990; Milner and Tofte, 1991]; a comparative discussion can be found in [Gunter, 1993].

Let us now look at such a semantics for PCF. It is given by a binary relation $\Downarrow$ between closed terms of the calculus. The binary relation is defined as the least relation that satisfies the axioms and rules in Table 8. In the description of these rules, the terms that appear on the right side have been written using the letters U, V, W for values rather than L, M, N for arbitrary terms. To read the rules, assume at first that U, V, W range over all terms. It can then be proved by an induction on the height of a derivation that if $M \Downarrow V$ for any terms M and V, then V is a term generated by the grammar 4.1. In other words, if rules such as

$$\frac{M \Downarrow \mathbf{succ}(V)}{\mathbf{pred}(M) \Downarrow V}$$

were instead written in the form

$$\frac{M \Downarrow \mathbf{succ}(N)}{\mathbf{pred}(M) \Downarrow N}$$

then it would be possible to *prove* that N has the form of a value V. It

Table 8. Natural Rules for Call-by-Name Evaluation of PCF

$$\mathbf{0} \Downarrow \mathbf{0} \qquad \mathbf{true} \Downarrow \mathbf{true} \qquad \mathbf{false} \Downarrow \mathbf{false}$$

$$\frac{M \Downarrow \mathbf{0}}{\mathbf{pred}(M) \Downarrow \mathbf{0}} \qquad \frac{M \Downarrow \mathbf{succ}(V)}{\mathbf{pred}(M) \Downarrow V} \qquad \frac{M \Downarrow V}{\mathbf{succ}(M) \Downarrow \mathbf{succ}(V)}$$

$$\frac{M \Downarrow \mathbf{0}}{\mathbf{zero?}(M) \Downarrow \mathbf{true}} \qquad \frac{M \Downarrow \mathbf{succ}(V)}{\mathbf{zero?}(M) \Downarrow \mathbf{false}}$$

$$\lambda x : s.\ M \Downarrow \lambda x : s.\ M \qquad \frac{M \Downarrow \lambda x : s.\ M' \qquad [N/x]M' \Downarrow V}{M(N) \Downarrow V}$$

$$\frac{M_1 \Downarrow \mathbf{true} \qquad M_2 \Downarrow V}{\mathbf{if}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3 \Downarrow V} \qquad \frac{M_1 \Downarrow \mathbf{false} \qquad M_3 \Downarrow V}{\mathbf{if}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3 \Downarrow V}$$

$$\frac{[\mu x : t.\ M/x]M \Downarrow V}{\mu x : t.\ M \Downarrow V}$$

is not hard to check that if $M \Downarrow U$ and $M \Downarrow V$, then $U \equiv V$, so $\Downarrow$ is a partial function. It is, moreover, possible to show that this gives the same semantics for PCF as the SOS:

Theorem 4.2.2. *$M \Downarrow V$ if, and only if, $M \rightarrow^* V$.*

4.3 Operational equivalence.

The question, now, is what these various formulations of the operational semantics of PCF have to do with the *types* for the system. Let us assume the perspective of the previous section and consider the question of what the interpretations of the PCF types should be. Interpreting them as sets as we did for the simply-typed λ-calculus leads to problems, however, when we wish to interpret recursion. The best-known approach to resolving this difficulty is to impose additional structure on the interpretations of types by using certain kinds of ordered sets, ordinarily known as *domains.* Assuming that we have found such a semantics—one that interprets a term M of type t as an element of the interpretation of t (modulo the values of free variables of M)—the key issue is the relationship between this form of semantics and the operational semantics of the language as described above. That is, we need the analogs to the soundness and completeness theorems given earlier, but now these results should be relative to an operational semantics rather than an equational theory. The trick is, in effect, to generate an equational theory from the operational semantics and study these properties relative to it. More precisely, this is done relative to a pre-order imposed on terms; this pre-order induces the desired equations.

A PCF *context* C is essentially a PCF term with a missing subterm marked by a place-holder [ ]. The PCF term obtained by filling the 'hole' in the term C by a term M is denoted $C[M]$. This is similar to substitution, but the placement of M into context C permits free variables of M to be bound within variable scopes determined by C. (In particular, contexts, unlike terms, are not considered equivalent modulo renaming of bound variables.) Evaluation contexts are a special class of contexts in which the missing subterm is in a special position. We define the key notion of operational equivalence as follows. Suppose M and N are terms of type t (that is, $H \vdash M : t$ and $H \vdash N : t$ for some H). Say M is an operational approximation of N and write $M \sqsubseteq_o N$ if, for every context C such that $C[M]$ and $C[N]$ are closed terms of ground type,

$$C[M] \Downarrow V \text{ implies } C[N] \Downarrow V.$$

It is possible to prove that $\sqsubseteq_o$ is a pre-order; terms M, N are *operationally equivalent,* and we write $M \approx N$ if $M \sqsubseteq_o N$ and $N \sqsubseteq_o M$

A semantics $[\![\cdot]\!]$ is said to be *adequate* if $[\![H \rhd M : t]\!] = [\![H \rhd N : t]\!]$ implies $N \approx M$. It is said to be *fully abstract* if it is sound and, $N \approx M$ implies $[\![H \rhd M : t]\!] = [\![H \rhd N : t]\!]$. Two well-known adequate semantics for PCF are the *bc-domains* interpretation $\mathcal{C}[\![\cdot]\!]$ and the *dI-domains* interpretation $\mathcal{D}[\![\cdot]\!]$. To describe each of these briefly some knowledge of domain theory will be assumed: some definitions are given below—further background can be found in [Abramsky and Jung, 1994].

4.4 bc-domains and dI-domains.

A *bc-domain* is an algebraic complete partial order that is bounded complete, that is, every subset that has an upper bound has a least upper bound (lub). If D and E are bc-domains, then the space of continuous functions $[D \to E]$ under the pointwise order is also a bc-domain. Such domains can be used to model PCF types by interpreting ground type expressions **num** and **bool** as the flat cpo's $\mathbb{N}_\perp$ (numbers together with least element $\perp$) and $\mathbb{T}$ (truth values **true**, **false** together with $\perp$) respectively and the higher types by taking $\mathcal{C}[\![s \to t]\!]$ to be $[\mathcal{C}[\![s]\!] \to \mathcal{C}[\![t]\!]]$. As before, meanings are defined on triples H, M, t where $H \vdash M : t$. An H-environment ρ is a partial function that assigns to each variable x such that $x \in H$ a value

$\rho(x)$ in $\mathcal{C}[\![H(x)]\!]$. The meaning $\mathcal{C}[\![H \rhd M : t]\!]$ is a function that assigns to each H-environment ρ a value

$$\mathcal{C}[\![H \rhd M : t]\!]\rho \in \mathcal{C}[\![t]\!].$$

The definition of this function follows the structure of the expression M (or, equivalently, the proof that $H \vdash M : t$). For the simply-typed λ-calculus fragment of PCF, the interpretation looks the same as before. The arithmetic and conditional expressions have a straight-forward interpretation. It is the interpretation of recursive functions that takes advantage of the additional structure of *domains*—as compared to *sets*—in the interpretation of types:

$$\mathcal{C}[\![H \rhd \mu x : t.\ M : t]\!]\rho = \mathbf{fix}(d \mapsto \mathcal{C}[\![H,\ x : t \rhd M : t]\!]\rho[x \mapsto d])$$

where **fix** is a function that gives the least fixed point of a continuous function. To show that the definition makes sense, one proves the following:

Lemma 4.4.1. *If $H' = H, x : s$ is a type assignment such that $H' \vdash M : t$, then the function*

$$d \mapsto \mathcal{C}[\![H' \rhd M : t]\!]\rho[x \mapsto d]$$

is continuous for any H'-environment ρ.

The following property is also easy to establish by induction on the height of a derivation tree:

Proposition 4.4.2. *If $M \to N$, then $\mathcal{C}[\![M]\!] = \mathcal{C}[\![N]\!]$.*

Proving that adequacy holds for $\mathcal{C}[\![\cdot]\!]$ is somewhat harder and beyond the scope of this chapter.

Another interpretation of PCF can be obtained by using bc-domains that are distributive and have property I. To define these properties precisely, let $\sqcup$ and $\sqcap$ stand for the least upper bound and greatest lower bound operators respectively.

Definition 4.4.3. A bc-domain D is said to be *distributive* if $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$ whenever $\{x, y\}$ has an upper bound. An algebraic cpo D *has property I* if $\{x \mid x \sqsubseteq a\}$ is finite for each compact element a of D. A distributive bc-domain that satisfies property I is called a *dI-domain.*

Interpreting the types of PCF as dI-domains requires one more crucial idea. It is not hard to find dI-domains D, E with the property that the continuous function space $[D \to E]$ under the pointwise order is *not* a dI-domain. Higher types for a model based on dI-domains cannot be interpreted with this construct. The key idea is given in the following:

Definition 4.4.4. A continuous function $f : D \to E$ between dI-domains D and E is *stable* if $f(x \sqcap y) = f(x) \sqcap f(y)$ whenever $\{x, y\}$ has an upper bound. If $f, g : D \to E$ are stable, then f is below g in the *stable ordering* and we write $f \sqsubseteq_s g$ if

$$x \sqsubseteq y \text{ implies } f(x) = f(y) \sqcap g(x)$$

for each $x, y \in D$.

It is possible to show that if D, E are dI-domains, then the poset of stable functions $[D \to_s E]$ under the stable ordering is also a dI-domain. The dI-domains can be used to give a semantics $\mathcal{D}[\![\cdot]\!]$ for PCF in basically the same way that bc-domains were used to give a semantics $\mathcal{C}[\![\cdot]\!]$ before. One must prove the analog of Lemma 4.4.1:

Lemma 4.4.5. *If $H, x : s \vdash M : t$, then the function*

$$d \mapsto \mathcal{D}[\![H, x : s \rhd M : t]\!]\rho[x \mapsto d]$$

is stable.

4.5 Full abstraction.

The use of algebraic cpo's and bounded completeness as a model of λ-calculus was developed by Dana Scott [Scott, 1976; Scott, 1981; Scott, 1982a; Scott, 1982b; Gunter and Scott, 1990]. The dI-domains were introduced by Gerard Berry [Berry, 1978; Berry, 1979; Berry *et al.*, 1985] in an effort to find a fully abstract model of PCF after Gordon Plotkin [Plotkin, 1976] demonstrated that the bc-domains model of PCF is *not* fully abstract. Plotkin proved this failure directly from the operational semantics of PCF, but the result can also be obtained semantically by using dI-domains as a 'non-standard' model. To do this, we need to demonstrate two terms that have the same operational behavior in all ground contexts but fail to be equal in the model. To this end, let

$$T, F : (\mathbf{bool} \to (\mathbf{bool} \to \mathbf{bool})) \to \mathbf{bool}$$

be the PCF terms given in Table 9. In the table there, Ω is a divergent program of boolean type (for instance $\mu x : \mathbf{bool}.\ x$ will do). The term F is the same as T except for the occurrence **false** in the fifth line.

Now, the programs T and F have the same operational behavior in all ground contexts. To see this, it suffices, by adequacy for the dI-domains model, to show that $\mathcal{D}[\![T]\!] = \mathcal{D}[\![F]\!]$. We show, in fact, that $\mathcal{D}[\![T]\!](\emptyset) = \mathcal{D}[\![F]\!](\emptyset) : f \mapsto \bot$, where $\emptyset$ is the 'arid' environment (which makes no

Table 9. Operationally Equivalent Programs with Different Denotations

```
$T \equiv \lambda f : \mathbf{bool} \to (\mathbf{bool} \to \mathbf{bool}).$
        $\mathbf{if}\ f(\mathbf{true})(\Omega)\ \mathbf{then}$
            $\mathbf{if}\ f(\Omega)(\mathbf{true})\ \mathbf{then}$
                $\mathbf{if}\ f(\mathbf{false})(\mathbf{false})\ \mathbf{then}\ \Omega$
                $\mathbf{else\ true}$
            $\mathbf{else}\ \Omega$
        $\mathbf{else}\ \Omega$

$F \equiv \lambda f : \mathbf{bool} \to (\mathbf{bool} \to \mathbf{bool}).$
        $\mathbf{if}\ f(\mathbf{true})(\Omega)\ \mathbf{then}$
            $\mathbf{if}\ f(\Omega)(\mathbf{true})\ \mathbf{then}$
                $\mathbf{if}\ f(\mathbf{false})(\mathbf{false})\ \mathbf{then}\ \Omega$
                $\mathbf{else\ false}$
            $\mathbf{else}\ \Omega$
        $\mathbf{else}\ \Omega$
```

assignments). To this end, suppose $\mathcal{D}[\![T]\!](\emptyset)(f) \neq \bot$. This can only happen if $\mathcal{D}[\![T]\!](\emptyset)(f) = \mathbf{true}$. If $f'(x, y) = f(x)(y)$ is the 'uncurrying' of f, then

$$\begin{aligned} f(\bot)(\bot) &= f'(\bot, \bot) = f'((\mathbf{true}, \bot) \sqcap (\bot, \mathbf{true})) \\ &= f'(\mathbf{true}, \bot) \sqcap f'(\bot, \mathbf{true}) = \mathbf{true} \end{aligned}$$

since f' is stable. On the other hand, $f(\mathbf{false})(\mathbf{false}) = f'(\mathbf{false}, \mathbf{false}) = \mathbf{false}$, and this contradicts the monotonicity of f! A similar argument applies to F, so we can conclude that the terms T and F have the same operational behavior. Indeed, they both have the same operational behavior as $\lambda x.\ \Omega$. Switching now to the interpretations of these terms in the bc-domain semantics, we can show that $\mathcal{C}[\![T]\!] \neq \mathcal{C}[\![F]\!]$. To do this, consider a function $\mathbf{por} : \mathbb{T} \to [\mathbb{T} \to \mathbb{T}]$, called the *parallel or,* defined by the left truth table in Table 10 where the values in the left column are those of the first argument and the values in the top row are those of the second argument. This can be contrasted with the truth table for the (left-to-right) sequential or defined by

$$\mathbf{or} = \mathcal{C}[\![\lambda x : \mathbf{bool}.\ \lambda y : \mathbf{bool}.\ \mathbf{if}\ x\ \mathbf{then\ true\ else}\ y]\!](\emptyset).$$

Note the difference in the value of $\mathbf{or}(\bot)(\mathbf{true})$ in the truth table for $\mathbf{or}$ given in Table 10. Now, the function $\mathbf{por}$ is monotone on a finite domain

Table 10. Truth Tables for Parallel and Sequential Disjunction

por	**true**	**false**	$\perp$
true	**true**	**true**	**true**
false	**true**	**false**	$\perp$
$\perp$	**true**	$\perp$	$\perp$

or	**true**	**false**	$\perp$
true	**true**	**true**	**true**
false	**true**	**false**	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$

and therefore continuous, so it is an element of the interpretation $\mathcal{C}[\![\mathbf{bool} \to (\mathbf{bool} \to \mathbf{bool})]\!]$. Hence

$$\mathcal{C}[\![T]\!](\emptyset)(\mathbf{por}) = \mathbf{true} \neq \mathbf{false} = \mathcal{C}[\![F]\!](\emptyset)(\mathbf{por})$$

so $\mathcal{C}[\![T]\!] \neq \mathcal{C}[\![F]\!]$, and $\mathcal{C}[\![\cdot]\!]$ is therefore not fully abstract.

It was shown by Berry that the dI-domains are *also* not fully abstract. Here is a brief semantic proof using the bc-domains model as a 'non-standard' interpretation. We show that two terms with different meanings in the dI-domains model have the same operational behavior. To this end, define monotone functions p, q on the truth value poset by taking $p : x \mapsto \mathbf{true}$ and taking q to be the function

$$q(x) = \begin{cases} \perp & \text{if } x = \perp \\ \mathbf{true} & \text{otherwise.} \end{cases}$$

The function q is below p in the pointwise order, but these two functions are unrelated in the stable order. Noting this, it is possible to see that the following function is an element of $\mathcal{D}[\![(\mathbf{bool} \to \mathbf{bool}) \to \mathbf{bool}]\!]$:

$$r(x) = \begin{cases} \mathbf{true} & \text{if } x = q \\ \mathbf{false} & \text{if } x = p \\ \perp & \text{otherwise.} \end{cases}$$

Now consider the programs

$$\begin{aligned} M &\equiv \lambda f.\ \mathbf{if}\ f(q)\ \mathbf{then}\ (\mathbf{if}\ f(p)\ \mathbf{then}\ \Omega\ \mathbf{else}\ \mathbf{true})\ \mathbf{else}\ \Omega \\ N &\equiv \lambda f.\ \mathbf{if}\ f(q)\ \mathbf{then}\ (\mathbf{if}\ f(p)\ \mathbf{then}\ \Omega\ \mathbf{else}\ \mathbf{false})\ \mathbf{else}\ \Omega \end{aligned}$$

where type tags have been dropped to reduce clutter. Clearly $\mathcal{D}[\![M]\!](r) \neq \mathcal{D}[\![N]\!](r)$. However, M and N have the same operational behavior because $\mathcal{C}[\![M]\!] = \mathcal{C}[\![N]\!]$. To see why this latter equation holds, just note that if $f(q) = \mathbf{true}$ for any function f that is monotone over the pointwise-ordered monotone functions of type $\mathbf{bool} \to \mathbf{bool}$, then $f(p) = \mathbf{true}$ as well.

The problem of finding a fully abstract model of PCF has a long history. But rather than ask whether PCF has such a model, one can also ask whether there is any extension of PCF that is fully abstract with respect, say, to the bc-domains semantics. Plotkin [Plotkin, 1976] showed that this is the case; indeed, the language PCF is fully abstract for the bc-domains model if one adds a primitive for computing the parallel disjunction **por** [Stoughton, 1991]. Interestingly, no similar way of extending the language seems to work for the dI-domains model [Jim and Meyer, 1991]. More details on full abstraction and related topics can be found in [Gunter, 1992]. Some current research directions are indicated in the concluding section of this chapter.

5 Types as Invariants

In Section 4 the idea of presenting an operational semantics for PCF was described. The semantics of types was given in terms of domains and then this interpretation was related to the equivalence induced by the operational semantics. But there is another basic relationship that one can show between the operational semantics and the types of the language based on the simple idea that the types are *invariants* of the operational semantics. This is a property known as *subject reduction.* For PCF and the SOS given in Table 7 it can be expressed as follows:

Theorem 5.0.1. *(Subject Reduction) If $H \vdash M : t$ and $M \rightarrow N$, then $H \vdash N : t$.*

By Theorem 4.2.2 this has the following:

Corollary 5.0.2. *If $H \vdash M : t$ and $M \Downarrow N$, then $H \vdash N : t$.*

This form of theorem is very useful for proving that certain kinds of runtime errors are avoided by programs that are type correct.

5.1 Runtime safety.

What properties are expected for the evaluation of a type-correct program beyond those that may hold of an arbitrary one? To appreciate the significance of the types, look again at the operational rules in Table 8. Take a typical rule such as the one for application in call-by-name:

$$\frac{M \Downarrow \lambda x : t.\ M' \qquad [N/x]M' \Downarrow V}{M(N) \Downarrow V}$$

This is the only rule whose conclusion describes how to derive a value for an application, so any attempt to prove that $M(N)$ has value V must use it. The rule requires that two hypotheses be established. Let us focus on the first. It was remarked before that if $M \Downarrow U$ for some value U, then U is the unique value that satisfies this relationship. Hence there are three possibilities that could result from the attempt to find a value for M:

1. there is no value U such that $M \Downarrow U$, or
2. there is a term M' such that $M \Downarrow \lambda x : t.\ M'$, or
3. there is a term U such that $M \Downarrow U$, but U does not have the form $\lambda x : t.\ M'$.

The first of these might occur because of divergence, or perhaps for some other reason. The second is the conclusion we must reach to find a value for $M(N)$. The third case arises in an 'abnormal' situation in which something other than an abstraction is being applied to an argument. For example, this would happen if we attempted to evaluate the application $\mathbf{0}(\mathbf{0})$ of the number $\mathbf{0}$ to itself. Here is what the type-correctness of $M(N)$ ensures: the third possibility above never occurs. In the example $\mathbf{0}(\mathbf{0})$ this is clear because $\mathbf{0}$ has type $\mathbf{num}$ rather than type $\mathbf{num} \to t$ as it would be required to have if it is to be applied to a number.

Although the first and third cases above both mean that $M(N)$ does not have a value, there is an important difference in the way this failure occurs. In particular, if it is found that $M \Downarrow U$ but U does not have the desired form, then it is possible to report immediately that the attempt to find a value for $M(N)$ has *failed.* This will not always be possible for the first case, since the failure to find a value for M may be due to an infinite regression of attempts to apply operational rules (this is what would happen for the term $\mu x : \mathbf{num}.\ x$ for example). Any attempt to determine whether this is the case through an effective procedure will fail, because this is tantamount to solving the halting problem. Hence, the last case is special.

For some guidance, let us consider the difference between these possibilities in a programming language. Here is an example of a Scheme program that will diverge when applied to an argument:

```
(define (f x) (f x))
```

Evaluating `(f 0)` in the read-eval-print loop will be a boring and unfulfilling activity that will probably be ended by an interruption by the programmer. This program diverges and therefore does not have a value. On the other hand, what happens if we attempt to evaluate the program `(0 0)` in the read-eval-print loop? There is no value for this expression, but we receive an instant warning of this limitation that may look like this:

```
Application of inapplicable object 0
```

Table 11. Operational Rules for Type Errors

$$\mathbf{tyerr} \Downarrow \mathbf{tyerr} \qquad \frac{L \Downarrow V \qquad V \notin \text{Boolean}}{\mathbf{if}\ L\ \mathbf{then}\ M\ \mathbf{else}\ N \Downarrow \mathbf{tyerr}}$$

$$\frac{M \Downarrow V \qquad V \notin \text{Number}}{\mathbf{pred}(M) \Downarrow \mathbf{tyerr}} \qquad \frac{M \Downarrow V \qquad V \notin \text{Number}}{\mathbf{succ}(M) \Downarrow \mathbf{tyerr}}$$

$$\frac{M \Downarrow V \qquad V \notin \text{Number}}{\mathbf{zero?}(M) \Downarrow \mathbf{tyerr}} \qquad \frac{M \Downarrow V \qquad V \notin \text{Lambda}}{M(N) \Downarrow \mathbf{tyerr}}$$

The difference between these two outcomes arises from the distinction between *divergence* and a *runtime type error.* While divergence is generally undetectable, the runtime type error can be reported when it arises.

To study these ideas rigorously, let us focus on a specific language. The following grammar defines the syntax of type expressions t and terms M of a calculus called PCF *with type errors.* The extended calculus is the same as PCF except for the inclusion of a new constant called **tyerr**. Here is the expanded grammar:

$$\begin{array}{lcl} t & ::= & \mathbf{num} \mid \mathbf{bool} \mid t \to t \\ M & ::= & \mathbf{tyerr} \mid \mathbf{0} \mid \mathbf{succ}(M) \mid \mathbf{pred}(M) \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{zero?}(M) \mid \\ & & x \mid \lambda x : t.\ M \mid MM \mid \mu x : t.\ M \mid \mathbf{if}\ M\ \mathbf{then}\ M\ \mathbf{else}\ M \end{array}$$

The typing rules for extended PCF are the same as those for PCF itself. Note, in particular, that the relation $H \vdash \mathbf{tyerr} : t$ fails for each H and t since there is no typing rule for proving such a relation.

The rules for a natural operational semantics are those given earlier in Table 8 together with the 'error' rules given in Table 11. Values in the new language are those of PCF together with the term **tyerr** for a type error. The rules in the table are defined using syntactic judgements such as $V \notin$ Boolean where Boolean is the set of values V such that $\vdash V :$ Boolean. In the rules, Number is the set of values of numerical type (that is, numerals) and Lambda those of higher type (that is, abstractions). The expanded set of rules has properties similar to those of the system without explicit error elements. For example, the relation $\Downarrow$ for the full language is still a partial function, that is, if $M \Downarrow U$ and $M \Downarrow V$, then $U \equiv V$. What differentiates this system from the previous one is the fact that the hypotheses of the rules now cover all possible patterns for the outcome of an evaluation. If the evaluation of a term calls for the evaluation of other terms, then the error rules indicate what is to be done if the result of evaluating these other terms is a type error or yields a conclusion having the wrong form.

Now, we would like to prove a theorem that says that the evaluation of a well-typed term does not produce a type error. Here is a more precise statement:

Theorem 5.1.1. *If $H \vdash M : t$ then it is not the case that $M \Downarrow$ **tyerr**.*

The result we need to show absence of runtime type errors can be proved using a form of subject reduction theorem. In this case the result must be proved relative to the typing system for PCF and the evaluation relation $\Downarrow$ defined as the least relation satisfying the rules in Table 11 as well as those in Table 8.

Theorem 5.1.2 (Subject Reduction). *Let M be a term in extended* **PCF.** *If $M \Downarrow V$ and $H \vdash M : t$, then $H \vdash V : t$.*

Proof. The proof is by induction on the height of the evaluation $M \Downarrow V$. I will do only the cases for predecessor, application, and recursion.

Case $M \equiv \mathbf{pred}(M')$. There are three possibilities for the last step in the derivation of $M \Downarrow V$. If $M' \Downarrow V'$, then we employ the induction hypothesis to conclude that $H \vdash V' : \mathbf{num}$. In particular, this means that $V' \in$ Number so $V' \equiv \mathbf{0}$ or the last step of the derivation must be an application of the rule

$$\frac{M \Downarrow \mathbf{succ}(V) \qquad V \in \text{Number}}{\mathbf{pred}(M) \Downarrow V}$$

where $V' \equiv \mathbf{succ}(V)$. If $V' \equiv \mathbf{0}$, then the desired conclusion is immediate, since $H \vdash \mathbf{0} : \mathbf{num}$. On the other hand, the only way to have $H \vdash \mathbf{succ}(V) : \mathbf{num}$ is if $H \vdash V : \mathbf{num}$, so this possibility also leads to the desired conclusion.

Case $M \equiv L(N)$. Say $H \vdash L : r \rightarrow s$ and $H \vdash N : r$. There are two operational rules that may apply to the evaluation of an application. The error rule in Table 7.3 could not apply to M, however, because of the inductive hypotheses on L. Hence the last step in the evaluation of M must have the following form:

$$\frac{L \Downarrow \lambda x : r.\ L' \qquad [N/x]L' \Downarrow V}{L(N) \Downarrow V}$$

Now, by the induction hypothesis, $H \vdash \lambda x : r.\ L' : r \rightarrow s$ so it must be that $H,\ x : r \vdash L' : s$. Hence, by Lemma 4.1.1(2), $H \vdash [N/x]L' : s$, and it therefore follows from the induction hypothesis that $H \vdash V : t$.

Case $M \equiv \mu x : t.\ M'$. In this case, $[\mu x : t.\ M'/x]M' \Downarrow V$. By Lemma 4.1.1, $H \vdash [\mu x : t.\ M'/x]M' : t$ so $H \vdash V : t$ by the induction hypothesis. ∎

Theorem 5.1.1 now follows immediately, since **tyerr** does not have a type.

The interest of Theorem 5.1.1, which is intended to assert that the evaluation of a well-typed program does not yield a type error, depends entirely on the nature of the error rules. Hence it is important to examine them closely to see that they do indeed encode all of the circumstances under which one would expect a type error to be reported. One problem with this way of asserting freedom from runtime errors is that mistakenly omitting a rule from Table 11 would make Theorem 5.1.1 easier to prove! Another way to express freedom from runtime errors is to use an SOS for the language and assert the result as a guarantee that computation does not get 'stuck' at a non-value. Here is such an assertion for PCF:

Theorem 5.1.3. *If $H \vdash M : t$ and M is not a value, then $M \to N$ for some N.*

5.2 Implicit types.

If having a type is viewed primarily as a property of a program that ensures desirable runtime behavior, then it may be a convenience if type-correctness is inferred automatically and programmers are relieved as far as possible of the need to to write type annotations. This leads us to the idea of an *implicit* type, that is, one not explicitly given as part of the program. Since a program without explicit types naturally provides less type information than one that has them, a key technical issue arises for such programs: while our discussion has been focused entirely on languages for which the type of a term, if it has one, is uniquely determined because of type tags, this property must be viewed differently when type tags are omitted. Let us now consider the λ-calculus *without* the type tags. Since the tags gave the types explicitly before, the new system is called the *implicitly*-typed (simply-typed) calculus. The syntax for the language is simply

$$M \ ::= \ x \mid \lambda x.\ M \mid MM$$

and the various syntactic conventions are exactly the ones used earlier for the simply-typed λ-calculus with type tags. The typing rules for the implicit system are almost the same as those for the explicit calculus, but the abstraction rule now has different properties. The rules are given in Table 12.

Let us consider now how one could find a type for a term with respect to the rules in Table 12, recalling that it was shown earlier that $[\mathrm{Abs}]^{-}$ leads to the failure of Lemma 3.1.2. If M is a term of the form $L(N)$, then types need to be found for L and N. If M has the form $\lambda x.\ M'$, then a type needs to be found for M' assuming that x has *some* type s. Now, in the case that we are looking for the type of a term like $\lambda x.\ x$, then, for *any* type s, we can find a type with the following instances of the projection and application rule:

Table 12. Implicitly-Typed λ-Calculus

[Proj]	$\dfrac{x \in H}{H \vdash x : H(x)}$
$[\text{Abs}]^-$	$\dfrac{H,\ x : s \vdash M : t}{H \vdash \lambda x.\ M : s \to t}$
[Appl]	$\dfrac{H \vdash M : s \to t \qquad H \vdash N : s}{H \vdash M(N) : t}$

$$\frac{x : s \vdash x : s}{\vdash \lambda x.\ x : s \to s}$$

The choice of s here is arbitrary, but each of the types for $\lambda x.\ x$ must have the form $s \to s$. Indeed this form precisely characterizes what types *can* be the type of $\lambda x.\ x$. Let us now consider a slightly more interesting example; let

$$M \equiv \lambda x.\ \lambda f.\ f(x).$$

In a typing derivation ending with a type for M, the last two steps must have the following form:

$$\cfrac{\cfrac{x : t_1, f : t_2 \vdash f(x) : t_3}{x : t_1 \vdash \lambda f.\ f(x) : t_2 \to t_3}}{\vdash M : t_1 \to (t_2 \to t_3)}$$

Letting H be the assignment $x : t_1, f : t_2$, the derivation of the hypothesis at the top must have the form

$$\frac{H \vdash x : t_1 \qquad H \vdash f : t_2}{x : t_1, f : t_2 \vdash f(x) : t_3}$$

where, to match the application rule, it must be the case that t_2 have the form $t_1 \to t_3$. It is not hard to see that *any* choice of t_1, t_2, t_3 satisfying this one condition will be derivable as a type for M. In short, the types that M can have are exactly characterized as those of the form

$$r \to (r \to s) \to s. \tag{5.1}$$

This suggests that there may be a way to reconstruct a general form for the type of a term in the implicit calculus. If the types r and s could be viewed as variables in the type 5.1, then we could say that a type t satisfies $\vdash M : t$ just in case t is a substitution instance of 5.1. What is most important though is the prospect that a type for a term could be determined even though the type tags are missing. For a calculus like PCF, this might make it possible to omit type tags and still ensure the kind of security asserted for the well-typed terms of the language in Theorem 5.1.1.

Of course, there are terms that cannot be given a type because they 'make no sense'. This is easy to see in PCF where there are constants: a term such as $\mathbf{0}(\mathbf{0})$ is clearly meaningless. In the λ-calculus by itself, however, there is a gray area between what is and what is not a sensible term. The implicit system of Table 12 judges that $\vdash M : t$ if, and only if, there is a term of the *explicitly*-typed λ-calculus of Table 4 from which M can be obtained by erasing the tags. For example, it is impossible to find a type for $\lambda f.\ f(f)$ with the implicit typing system. To see this, suppose on the contrary that this term does have a type. The derivation of the type must end with an instance of [Abs]$^-$:

$$\frac{f : s \vdash f(f) : t}{\vdash \lambda f.\ f(f) : s \to t}$$

The proof of the hypothesis must be an instance of [Appl]:

$$\frac{f : s \vdash f : u \to t \qquad f : s \vdash f : u}{f : s \vdash f(f) : t},$$

which, by the axiom [Proj], means that $u \to t \equiv s \equiv u$. However, there is no type that has this property since the type u cannot have itself as a proper subterm. But there are contexts in which this term seems to make some sense. For example, it might be argued that the term $(\lambda x.\ x(x))(\lambda y.\ y)$ is harmless, since the x in the first abstraction is bound to an argument in the application that can indeed be applied to itself.

The calculus ML_0, which we now introduce, has a type system that can be viewed as a compromise between the implicit type discipline of Table 12 (which is essentially the simply-typed λ-calculus) and the *untyped* λ-calculus for which no typing system is used. ML_0 is a core representation of the system of the ML programming language. The goal is to provide some level of additional flexibility to the implicit typing discipline while maintaining a close link to the simply-typed λ-calculus. The key idea in the system is the inclusion of a syntax class of parameterized types. The full grammar for the language is given as follows:

$$\begin{array}{rcl} x & \in & \mathrm{TermVariable} \\ a & \in & \mathrm{TypeVariable} \\ t & ::= & a \mid t \to t \\ T & ::= & t \mid \Pi a.\ T \\ M & ::= & x \mid \lambda x.\ M \mid MM \mid \mathbf{let}\ x = M\ \mathbf{in}\ M \end{array}$$

In addition to the primitive syntax class of term variables x, a new syntax class of *type variables* a has been added. A *type scheme* T has the form

$\Pi a_1.\ \Pi a_2. \ldots \Pi a_n.\ t$ where the type variables $a_1, \ldots, a_n$ bind any occurrences of these variables in the type t. The usual rules for substitution and α-equivalence apply to type schemes: expressions are taken modulo α-equivalence, and a substitution should not result in any free variable of the substituted type becoming bound after the substitution. In other words, for any substitution σ,

$$\sigma(\Pi a_1. \ldots \Pi a_n.\ t) \equiv \Pi a_1. \ldots \Pi a_n.\ \sigma(t)$$

where it is implicitly assumed by the Bound Variable Convention that no a_i is in the support of σ and none has a free occurrence in $\sigma(b)$ for any b in the support of σ. We write $\mathrm{Ftv}(T)$ for the free type variables of a scheme T. The language includes one new construct for terms called a *let*. In an expression $L \equiv \mathbf{let}\ x = M\ \mathbf{in}\ N$, free occurrences of x in M are free in L while those that occur in N are bound by the **let**.

The typing rules for ML_0 will include a generalization of the projection rule in the implicit system that allows the type of a variable to be any type obtained by instantiating the Π-bound variables of a scheme associated with it in a type assignment.

Definition 5.2.1. A type s is said to be an *instance* of a type scheme $T \equiv \Pi a_1. \ldots \Pi a_n.\ t$ if there is a substitution σ with its support contained in $\{a_1, \ldots, a_n\}$ such that $\sigma(t) = s$. If s is an instance of T then we write $s \leq T$.

Assignments in ML_0 are defined similarly to assignments for simple types, but an ML_0 type assignment associates type *schemes* to term variables. Specifically, an *assignment* is a list H of pairs $x : T$ where x is a term variable and T is a type scheme. The set of free type variables $\mathrm{Ftv}(H)$ in an assignment H is the union of the sets $\mathrm{Ftv}(H(x))$ where $x \in H$. To give the typing rules for the system it is necessary to define a notion of the *closure* of a type relative to an assignment. This is a function on assignments H and types t such that

$$\mathrm{close}(H;\ t) = \Pi a_1. \ldots \Pi a_n.\ t$$

where $\{a_1, \ldots, a_n\} = \mathrm{Ftv}(t) - \mathrm{Ftv}(H)$. It is assumed that the function close chooses some particular order for the Π bindings here; it does not actually matter what this order is, but we can simply assume that our typing judgements are defined relative to a particular choice of the function close. A typing judgement is a triple $H \Vdash M : t$ where H is an assignment, M a term, and t a type. The typing rules for the system appear in Table 13. The symbol $\Vdash$ has been used in place of $\vdash$ for this system to distinguish it from the implicit system of Table 12 and from another system to which it will be compared later (the one in Table 20 to be precise). The rules

Table 13. Typing Rules for ML$_0$

[Proj]	$\dfrac{x : T \in H \qquad t \leq T}{H \Vdash x : t}$
[Abs]$^-$	$\dfrac{H,\ x : s \Vdash M : t}{H \Vdash \lambda x.\ M : s \to t}$
[Appl]	$\dfrac{H \Vdash M : s \to t \qquad H \Vdash N : s}{H \Vdash M(N) : t}$
[Let]	$\dfrac{H \Vdash M : s \qquad H, x : \mathrm{close}(H;\ s) \Vdash N : t}{H \Vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : t}$

for abstraction and application are the same as for the implicit typing system. The rule [Proj] for variables is different though because the type of a variable x can be any instance of the type scheme $H(x)$. The rule [Let] for the let construct gives the type of the let as that of N in an assignment where the type associated with x is the closure of the type of M. Note that there is a rule for each clause for a term in the grammar of the language, and the hypotheses of each rule are judgements about subterms of that term.

A basic property of the type variables and substitution in the system is given by the following:

Lemma 5.2.2. *If $H \Vdash M : t$, then $[s/a]H \Vdash M : [s/a]t$.*

In particular, if $a \notin \mathrm{Ftv}(H)$, then $[s/a]H \equiv H$, so $H \Vdash M : t$ implies $H \Vdash M : [s/a]t$.

As in the implicit simply-typed system, ML$_0$ does not have a type for the term $\lambda f.\ f(f)$. However, if f is let-bound to an appropriate value M, then the term

$$N \equiv \mathbf{let}\ f = M\ \mathbf{in}\ f(f)$$

can have a type. To see this in a specific example, take M to be the identity combinator $\lambda x.\ x$. Let a be a type variable, let us show that N has type $a \to a$. First of all,

$$\frac{x : a \Vdash x : a}{\Vdash \lambda x.\ x : a \to a}$$

follows from [Proj] and [Abs]$^-$. Now, by [Proj], we must also have the hypotheses of the following instance of [Abs]$^-$:

$$\frac{f : \Pi a.\ a \to a \Vdash f : a \to a \qquad f : \Pi a.\ a \to a \Vdash f : (a \to a) \to (a \to a)}{f : \Pi a.\ a \to a \Vdash f(f) : a \to a}$$

Note, in particular, that it is possible to instantiate the Π-bound variable a as the type $a \to a$ in one hypothesis and simply as a in the other. From the derivations above, we now have both hypotheses of this instance of the [Let] rule:

$$\frac{\Vdash \lambda x.\ x : a \to a \qquad f : \Pi a.\ a \to a \Vdash f(f) : a \to a}{\Vdash \mathbf{let}\ f = \lambda x.\ x\ \mathbf{in}\ f(f) : a \to a}$$

One of the most important characteristics of this typing system is the fact that we can determine whether a term has a type in a given assignment. Given an assignment H and a substitution σ, let $\sigma(H)$ be the assignment that associates $\sigma(H(x))$ to each $x \in H$. That is, if $H \equiv x_1 : T_1, \ldots, x_n : T_n$, then

$$\sigma(H) \equiv x_1 : \sigma(T_1), \ldots, x_n : \sigma(T_n).$$

Given assignment H and term M, define $\mathcal{S}(H, M)$ to be the set of all pairs (σ, t) such that σ is a substitution, t is a type, and $\sigma(H) \Vdash M : t$. There is an algorithm that, given H and M, provides an element of $\mathcal{S}(H, M)$ if the algorithm succeeds. To describe the algorithm, some background on substitutions is required.

Let σ and τ be substitutions. Then σ is said to be *more general* than τ if there is a substitution σ' such that $\sigma' \circ \sigma = \tau$. Given types s and t, a *unifier* for s, t is a substitution σ such that $\sigma(s) = \sigma(t)$. A *most general unifier* for s, t is a unifier σ that is more general than any other unifier for these types.

Theorem 5.2.3. *If there is a unifier for a pair of types, then there is also a most general unifier for them.*

This theorem and an algorithm for calculating most general unifiers was introduced in [Robinson, 1965]. The reason for introducing unifiers at this point is to to describe an algorithm of Robin Milner [Milner, 1978] called *algorithm* $\mathcal{W}$. It is given by induction on the structure of M by the following cases:

- Case $M \equiv x$. If $x \in H$, then the value is the identity substitution paired with the instantiation of the scheme $H(x)$ by a collection of fresh type variables. In other words, if $H(x) \equiv \Pi a_1 \ldots \Pi a_n.\ s$ where $a_1, \ldots, a_n$ are new type variables, then

  $$\mathcal{W}(H;\ x) = (\mathbf{id},\ s)$$

  where $\mathbf{id}$ is the identity map. If $x \notin H$, then the value of $\mathcal{W}(H;\ M)$ is failure.

- Case $M \equiv \lambda x.\ M'$. Suppose a is a new type variable and $(\sigma, t) = \mathcal{W}(H, x : a;\ M')$. Then

$$\mathcal{W}(H;\ \lambda x.\ M') = (\sigma,\ \sigma(a) \to t).$$

If, on the other hand, the value of $\mathcal{W}(H, x : a;\ M')$ is failure, then so is $\mathcal{W}(H;\ M)$.

- Case $M \equiv L(N)$. Suppose $(\sigma_1, t_1) = \mathcal{W}(H;\ L)$ and $(\sigma_2, t_2) = \mathcal{W}(\sigma_1(H);\ N)$. Let a be a new type variable. If there is a most general unifier σ for $\sigma_2(t_1)$ and $t_2 \to a$, then

$$\mathcal{W}(H;\ L(N)) = (\sigma \circ \sigma_2 \circ \sigma_1,\ \sigma a).$$

  In the event that there is no such unifier or if the value of $\mathcal{W}(H;\ L)$ or $\mathcal{W}(\sigma_1(H);\ N)$ is failure, then this is also the value of $\mathcal{W}(H;\ M)$.

- Case $M \equiv \textbf{let}\ x = L\ \textbf{in}\ N$. Suppose $(\sigma_1, s_1) = \mathcal{W}(H;\ L)$ and $H' \equiv \sigma_1(H),\ x : \mathrm{close}(\sigma_1(H);\ s_1)$. If $(\sigma_2, s_2) = \mathcal{W}(H';\ N)$, then

$$\mathcal{W}(H;\ M) = (\sigma_2 \circ \sigma_1,\ s_2).$$

  If, on the other hand, the value of $\mathcal{W}(H;\ L)$ or $\mathcal{W}(H';\ N)$ is failure, then that is also the value of $\mathcal{W}(H;\ M)$.

To prove that algorithm $\mathcal{W}$ is sound, it helps to have the following:

Lemma 5.2.4. *If $H, x : \Pi a_1. \ldots \Pi a_n.\ s \Vdash M : t$, then $H, \mathrm{close}(H;\ s) \Vdash M : t$.*

To understand this, note that the Bound Variable Convention insists that the variables a_i that appear in s are not in $\mathrm{Ftv}(H)$. The desired soundness property can be stated precisely as follows:

Theorem 5.2.5. *If $\mathcal{W}(H;\ M)$ exists, then it is an element of $\mathcal{S}(H;\ M)$.*

Proof. Suppose $\mathcal{W}(H;\ M)$ exists; we must show that $\sigma(H) \Vdash M : t$. The proof is by induction on the structure of M. If $M \equiv x$ is a variable, then t is an instantiation of $H(x)$. This means $H \Vdash x : t$ by the typing rule for variables.

Case $M \equiv \lambda x.\ M'$. If $(\sigma, t) = \mathcal{W}(H, x : a;\ M')$, then $\sigma(H, x : a) \Vdash M' : t$ by the inductive hypothesis. Thus $\sigma(H), x : \sigma(a) \Vdash M' : t$ so $\sigma(H) \Vdash \lambda x.\ M' : \sigma(a) \to t$ by the typing rule for abstraction. This means $\mathcal{W}(H;\ M) = (\sigma(H), \sigma(a) \to t) \in \mathcal{S}(H;\ M)$.

Case $M \equiv L(N)$. If $(\sigma_1, t_1) = \mathcal{W}(H;\ L)$ and $(\sigma_2, t_2) = \mathcal{W}(\sigma_1(H);\ N)$, then, by the inductive hypothesis, $\sigma_1(H) \Vdash L : t_1$ and $\sigma_2(\sigma_1(H)) \Vdash N : t_2$. Since $\mathcal{W}(H;\ M)$ exists, there is a substitution σ such that $\sigma(\sigma_2(t_1)) \equiv$

$\sigma(t_2 \to a) \equiv \sigma(t_2) \to \sigma(a)$. By Lemma 5.2.2, $\sigma \circ \sigma_2 \circ \sigma_1(H) \Vdash L : \sigma \circ \sigma_2(t_1)$ so

$$\sigma \circ \sigma_2 \circ \sigma_1(H) \Vdash L : \sigma(t_2) \to \sigma(a).$$

Also by the inductive hypothesis,

$$\sigma \circ \sigma_2 \circ \sigma_1(H) \Vdash N : \sigma(t_2).$$

Combining these facts with the rule for the typing of applications, we can conclude that

$$\sigma \circ \sigma_2 \circ \sigma_1(H) \Vdash L(N) : \sigma(a),$$

which means that $\mathcal{W}(H;\ M) \in \mathcal{S}(H;\ M)$.

Case $M \equiv$ **let** $x = L$ **in** N. If $(\sigma_1, s_1) = \mathcal{W}(H;\ L)$ and $(\sigma_2, s_2) = \mathcal{W}(H';\ N)$ where $H' = \sigma_1(H), x : \mathrm{close}(\sigma_1(H), s_1)$, then, by the inductive hypothesis, $\sigma_1(H) \Vdash L : s_1$ and $\sigma_2(H') \Vdash N : s_2$. By Lemma 5.2.2,

$$\sigma_2 \circ \sigma_1(H) \Vdash L : \sigma_2(s_1).$$

If $\Pi a_1 \ldots \Pi a_n.\ s_1 \equiv \mathrm{close}(\sigma_1(H), s_1)$, then

$$\sigma_2 \circ \sigma_1(H), x : \Pi a_1 \ldots \Pi a_n.\ \sigma_2(s_1) \Vdash N : s_2$$

so, by Lemma 5.2.4,

$$\sigma_2 \circ \sigma_1(H), x : \mathrm{close}(\sigma_2 \circ \sigma_1(H);\ \sigma_2(s_1) \Vdash N : s_2.$$

By the rule for typing lets, this says that $\mathcal{W}(H;\ M) = (\sigma_2 \circ \sigma_1, s_2) \in \mathcal{S}(H;\ M)$. ∎

Of course, this only proves that the answer calculated by $\mathcal{W}$ is 'sound'; another question, addressed in [Damas and Milner, 1982] led to the formulation of a theorem describing the sense in which the type inferred by algorithm $\mathcal{W}$ is the 'best' type possible. To be precise,

Definition 5.2.6. A *principal type* for H and M is a pair $(\sigma, s) \in \mathcal{S}(H, M)$ such that, for any other pair $(\tau, t) \in \mathcal{S}(H, M)$, there is a substitution σ' such that

- $\tau(H) = \sigma' \circ \sigma(H)$ and
- t is an instance of $\sigma'(\mathrm{close}(\sigma(H);\ s))$.

In the case that H is the empty assignment, this boils down to saying that if a closed term M has a type at all, then there is a type s such that M

has type s and, for any other type t, M has type t only if $t = \tau s$ for some substitution τ. Damas and Milner state the following:

Theorem 5.2.7. *If there is a type t such that $H \Vdash M : t$, then $\mathcal{W}(H;\ M)$ is a principal type scheme for H and M.*

Its proof appears in the thesis [Damas, 1985]. In practice there are many optimizations that can be done to provide a more efficient implementation of principal type scheme inference. One discussion of implementation has appears in [Cardelli, 1987]. Most books on ML include some discussion of ML type inference; for instance [Sokolowksi, 1991] provides code for unification and inference using the Standard ML module system.

5.3 Runtime safety for assignments and continuations.

To appreciate the value of Algorithm $\mathcal{W}$, it is necessary to consider it in connection with the way in which programs will be evaluated. This was not specified above, but this does not mean that the matter is trivial or unimportant; on the contrary, this topic requires some careful consideration if pitfalls are to be avoided. A straight-forward approach to the semantics of a term **let** $x = N$ **in** M is to treat it *operationally* as an application $(\lambda x.\ M)N$. The typing system of ML_0 treats these two terms differently, but it is possible to treat them as the same operationally. This is the approach taken by the Standard ML definition. It leads to complexities, however, when one considers the interaction between certain computational extensions and evaluation order. Let us now consider this issue and look at one possible solution proposed in [Leroy, 1993].

Let us assume that our language is to be given a call-by-value evaluation order and extended with the primitives and the rule for pairs that appear in Table 14. If desired, a sequencing operator can be included by taking $M; N$ to be syntactic sugar for $(\lambda x.\ N)M$ where x does not appear in N.

Algorithm $\mathcal{W}$ can easily be adapted to deal with these extra constructs. The new primitives can be treated as if they were free variables with the appropriate type schemes, and the inference for pairs can be done componentwise. Unfortunately this approach causes difficulties when taken with the usual evaluation order for the **let** construct, if this construct is viewed as syntactic sugar for an application as described above. A classic example of the difficulty with the use of Algorithm $\mathcal{W}$ for this language is given by the following program, which is written in a pseudo-code compromise between Standard ML and ML_0:

```
let r = ref(fn x => x)
 in update(r, fn x => x+1);
    (deref r)(true)
end
```

Table 14. Assignments, continuations, and pairs.

$\mathbf{ref} : \Pi a.\ a \to \mathbf{ref}(a)$
$\mathbf{deref} : \Pi a.\ \mathbf{ref}(a) \to a$
$\mathbf{update} : \Pi a.\ (\mathbf{ref}(a) \times a) \to a$
$\mathbf{callcc} : \Pi a.\ (\mathbf{cont}(a) \to a) \to a$
$\mathbf{throw} : \Pi a.\ \Pi b.\ (a \times \mathbf{cont}(a)) \to b$
$\mathbf{fst} : \Pi a.\ \Pi b.\ (a \times b) \to a$
$\mathbf{snd} : \Pi a.\ \Pi b.\ (a \times b) \to b$

$$\frac{H \vdash M : s \qquad H \vdash N : t}{H \vdash (M, N) : s \times t}$$

This program will pass Algorithm $\mathcal{W}$ because the type of r will be

$$\Pi a.\ \mathbf{ref}(a \to a).$$

Suppose the three primitives `ref`, `update`, and `deref` are treated as having the semantics of their Standard ML analogs `ref`, :=, and !, and the semantics of `let` is the one of SML. Then there will be a runtime error when r is dereferenced and an attempt is made to add `1` to `true`. The problem here is that the `update` operation on `r` instantiates the type of `r` so that the type of the dereferenced value is an inappropriate specialization of the polymorphic type of `r` at the point it is applied to `true`.

A similar problem arises with the addition of the control primitives `callcc` and `throw` to the language. Let us suppose that these primitives are given the semantics that `callcc` and the application of continuations have in Scheme. Duba, Harper, and MacQueen [Duba *et al.*, 1991] noted that Algorithm $\mathcal{W}$ may typecheck programs with these constructs that yield runtime errors. Here is an example from [Leroy, 1993]:

```
let later = callcc(fn k =>
                     (fn x => x,
                      fn f => throw(k, (f, fn g => 1))))
 in print(first(later)("Hello  World"));
    second(later)(fn x => x+1)
end
```

When invoked, the continuation `k` essentially carries out a reassignment of the identifier `later`. To see this and why it is a problem, let us trace the evaluation. After the `let` binding of `later`, the `first` coordinate of `later`—the identity function—will be invoked on a string, which is printed. Then the `second` coordinate is invoked on a function which is 'thrown' as the first coordinate of a pair to the previously captured continuation `k`. At this point, the evaluation proceeds in the same manner as the following program:

```
let later = (fn x => x+1, fn g => 1)
 in print(first(later)("Hello  World"));
    second(later)(fn x => x+1)
end
```

which leads to a type error when `first(later)` is applied because this leads to the addition of 1 and a string.

There have been a variety of proposals about how to deal with these problems by restricting the ML type system [Damas, 1985; Tofte, 1990]. An alternative is to change the evaluation order of the **let** expressions themselves. To illustrate this idea, let us consider an extension of ML_0 which we call ML_1. It has the following grammar:

$$\begin{array}{lcl}
x & \in & \text{TermVariable} \\
a & \in & \text{TypeVariable} \\
t & ::= & a \mid t \to t \mid t \times t \mid \mathbf{ref}(t) \mid \mathbf{cont}(t) \\
T & ::= & t \mid \Pi a.\, T \\
M & ::= & x \mid \lambda x.\, M \mid MM \mid \mathbf{let}\ x = M\ \mathbf{in}\ M \mid (M, M) \mid C \\
C & ::= & \mathbf{ref} \mid \mathbf{deref} \mid \mathbf{update} \mid \mathbf{callcc} \mid \mathbf{throw} \mid \mathbf{fst} \mid \mathbf{snd}
\end{array}$$

and its typing rules are those of ML_0 together with the types for the constants and pairs as given above.

The inclusion of assignments in the language leads us to incorporate notions of environment and store into our semantics, while the inclusion of first class continuations leads us to consider control as well. In earlier semantics the idea of a store could be ignored. Environments were avoided by using syntactic substitution to instantiate formal parameters to actual parameters, and control was expressed indirectly through the hypotheses of rules in SOS or natural semantics. Let us now consider an alternative way of describing a semantics which is essentially an *abstract machine*. The machine is described through a collection of transition rules in which environments, store, and control are all made explicit. The machine here is essentially a reformulation of the natural semantics appearing in [Leroy, 1993]. It closely resembles the abstract machine in [Felleisen and Friedman, 1986].

To describe it, we require some further concepts. The following grammar defines values V, continuations κ, and thunks T in terms of environments ρ and stores σ:

$$\begin{array}{lcl} l & \in & \text{Location} \\ V & ::= & \mathbf{Closure}(M,\rho) \mid (V,V) \mid l \mid \kappa \mid C \\ \kappa & ::= & \mathbf{Stop} \mid \mathbf{Apply1}(M,\rho,\kappa) \mid \mathbf{Apply2}(V,\kappa) \mid \\ & & \mathbf{Pair1}(M,\rho,\kappa) \mid \mathbf{Pair2}(V,\kappa) \\ T & ::= & \mathbf{Thunk}(M,\rho) \end{array}$$

An *environment* ρ is partial function with finite domain called an *environment* that maps a variable x to a value V or a thunk T. A store σ is a partial function with finite domain called a *store* that maps locations l to values V.

The machine is given in terms of a set of rewrite rules involving two operators push and pop. The operator push(M,ρ,σ,κ) takes a term M, an environment ρ, a store σ, and a continuation κ as its arguments. The operator pop(V,σ,κ) takes a value V, a store σ, and a continuation κ as arguments. If they terminate, the rewrite rules produce a pair (V,σ) or type error, **tyerr**, at the end. The relation $\rightarrow$ is defined to be the least one satisfying the rules in Table 15. We assume we are given a function **new** from stores to locations such that **new**(σ) is a location not in the domain of definition of σ. This function is used in the semantics of the **ref** constant for allocating new locations in memory. Intuitively, the push machine evaluates a term in stages, delaying parts of the calculation by pushing chores onto κ, which represents the continuation stack. When this evaluation reaches a value, the pop machine is invoked to consult the continuation stack to determine what should be done with the value.

The key result is the following:

Theorem 5.3.1. *If M is type correct, then it is not the case that*

$$\text{push}(M,\emptyset,\emptyset,\mathbf{Stop}) \rightarrow^* \mathbf{tyerr}.$$

The proof requires establishing a set of type invariants for environments, stores, and continuations. A further fact that can be shown is that, given a suitable formulation of types for values, if push$(M,\emptyset,\emptyset,\mathbf{Stop}) \rightarrow^* (V,\sigma)$, then the type of V in σ is the same as that of M.

Another approach to dealing with the semantics of **let** is to restrict the declaration so that polymorphism is only permitted in expressions that do not require any evaluation. This would make it illegal to write programs such as

```
let f = g o h in ...
```

where `o` is the higher-order composition function, but one could use an η-expansion and instead write

```
let f = fn x => g (h x) in ...
```

Andrew Wright, who introduced this idea [Wright, 1993], has been able to demonstrate its practicality for a number of substantial SML programs.

Table 15. Abstract Machine for ML_1.

$$\begin{array}{l}
\mathrm{push}(x,\rho,\sigma,\kappa) \to \mathrm{pop}(\rho(x),\sigma,\kappa) \qquad \text{if } \rho(x) \text{ is a value} \\
\mathrm{push}(x,\rho,\sigma,\kappa) \to \mathrm{push}(M,\rho',\sigma,\kappa) \qquad \text{if } \rho(x) = \mathbf{Thunk}(M,\rho') \\
\mathrm{push}(\lambda x.\ M,\rho,\sigma,\kappa) \to \mathrm{pop}(\mathbf{Closure}(\lambda x.\ M,\rho),\sigma,\kappa) \\
\mathrm{push}(MN,\rho,\sigma,\kappa) \to \mathrm{push}(M,\rho,\sigma,\mathbf{Apply1}(N,\rho,\kappa)) \\
\mathrm{push}(C,\rho,\sigma,\kappa) \to \mathrm{pop}(C,\sigma,\kappa) \\
\mathrm{push}((M,N),\rho,\sigma,\kappa) \to \mathrm{push}(M,\rho,\sigma,\mathbf{Pair1}(N,\rho,\kappa)) \\
\mathrm{push}(\mathbf{let}\ x = M\ \mathbf{in}\ N,\rho,\sigma,\kappa) \to \mathrm{push}(N,\rho[x \mapsto \mathbf{Thunk}(M,\rho)],\sigma,\kappa) \\
\mathrm{push}(M,\rho,\sigma,\kappa) \to \mathbf{tyerr} \qquad \text{in all other cases}
\end{array}$$

$$\begin{array}{l}
\mathrm{pop}(V,\sigma,\mathbf{Apply1}(N,\rho,\kappa)) \to \mathrm{push}(N,\rho,\sigma,\mathbf{Apply2}(V,\kappa)) \\
\mathrm{pop}(V,\sigma,\mathbf{Apply2}(\mathbf{Closure}(\lambda x.\ M,\rho),\kappa) \to \mathrm{push}(M,\rho[x \mapsto V],\sigma,\kappa) \\
\mathrm{pop}(V,\sigma,\mathbf{Apply2}(\mathbf{ref},\kappa)) \to \mathrm{pop}(\mathbf{new}(\sigma),\sigma[\mathbf{new}(\sigma) \mapsto V],\kappa) \\
\mathrm{pop}(l,\sigma,\mathbf{Apply2}(\mathbf{deref},\kappa)) \to \mathrm{pop}(\sigma(l),\sigma,\kappa) \\
\mathrm{pop}((l,V),\sigma,\mathbf{Apply2}(\mathbf{update},\kappa)) \to \mathrm{pop}(V,\sigma[l \mapsto V],\kappa) \\
\mathrm{pop}(\mathbf{Closure}(\lambda k.\ M,\rho),\sigma,\mathbf{Apply2}(\mathbf{callcc},\kappa)) \to \\
\quad \mathrm{push}(M,\rho[k \mapsto \kappa],\sigma,\kappa) \\
\mathrm{pop}((V,\kappa'),\sigma,\mathbf{Apply2}(\mathbf{throw},\kappa)) \to \mathrm{pop}(V,\sigma,\kappa') \\
\mathrm{pop}((U,V),\sigma,\mathbf{Apply2}(\mathbf{fst},\kappa)) \to \mathrm{pop}(U,\sigma,\kappa) \\
\mathrm{pop}((U,V),\sigma,\mathbf{Apply2}(\mathbf{snd},\kappa)) \to \mathrm{pop}(V,\sigma,\kappa) \\
\mathrm{pop}(U,\sigma,\mathbf{Pair1}(N,\rho,\kappa)) \to \mathrm{push}(N,\rho,\sigma,\mathbf{Pair2}(U,\kappa)) \\
\mathrm{pop}(V,\sigma,\mathbf{Pair2}(U,\kappa)) \to \mathrm{pop}((U,V),\sigma,\kappa) \\
\mathrm{pop}(V,\sigma,\mathbf{Stop}) \to (V,\sigma) \\
\mathrm{pop}(V,\sigma,\kappa) \to \mathbf{tyerr} \qquad \text{in all other cases}
\end{array}$$

6 Types as Subsets

So far we have modeled types as objects drawn from a collection of spaces or as syntactic invariants. Each closed, well-typed term denotes a value in the space that interprets its type or expresses a property that is unchanged by the evaluation of a term. Let us now consider another perspective in which a type is viewed as a subset of a *universe* of elements modeling terms. In this approach, the meaning $[\![M]\!]$ of a closed, well-typed term M is a member of the universe U, and $[\![M]\!]$ lies in the subset $[\![t]\!] \subseteq U$ of the universe that interprets the type t of M. There are several ways to view these subsets, resulting in different models. Our discussion begins with an examination of the interpretation of the *untyped* λ-calculus using a model of the simply-typed calculus that satisfies a special equation. We then consider how a model of the untyped calculus can be viewed as a model of the typed one by interpreting types as subsets.

6.1 Untyped λ-calculus.

The *untyped* λ-calculus is essentially the calculus obtained by removing the type system from the simply-typed calculus. The terms of the untyped calculus are generated by the following grammar:

$$M \ ::= \ x \mid MM \mid \lambda x.\ M$$

where the abstraction of the variable x over a term M binds the free variable occurrences of x in M. (These are the same terms used for the implicitly-typed system before.) Expressions such as $M', N, N_1, \ldots$ are used to range over untyped λ-terms as well as typed terms—context must determine which class of terms is intended. For discussions below relating typed and untyped calculi, P, Q, R are used for terms with type tags and L, M, N for those without such tags. Untyped λ-terms are subject to the same conventions about bound variables as we applied earlier to terms with type tags on bound variables. In particular, terms are considered equivalent if they are the same up to the renaming of bound variables (where no such renaming leads to capture of a variable by a new binding). The equational rules for the untyped *$\lambda\beta$-calculus* are given in Table 16. They are very similar to those of the typed calculus. The untyped *$\lambda\beta\eta$-calculus* is obtained by including the untyped version of the η-rule,

$$\{\eta\} \qquad \lambda x.\ M(x) = M,$$

where, as before, the variable x does not appear free in the expression M. It is not hard to see that every term of the simply-typed λ-calculus gives rise to a term of the untyped calculus obtained by 'erasing' the type tags on its free variables. More precisely, we define the *erasure* erase(P) of a term P of the simply-typed calculus by induction as follows:

$$\begin{aligned}
&\mathrm{erase}(x) \equiv x\\
&\mathrm{erase}(P(Q)) \equiv (\mathrm{erase}(P))(\mathrm{erase}(Q))\\
&\mathrm{erase}(\lambda x : t.\ P) \equiv \lambda x.\ \mathrm{erase}(P).
\end{aligned}$$

Although every term of the untyped calculus can be obtained as the erasure of a tagged one, it is not the case that every untyped term can be obtained as the erasure of a *well-typed* tagged term. For example, if erase$(P) \equiv \lambda x.\ x(x)$, then there is no context H and type expression t such that $H \vdash$

Table 16. Equational Rules for Untyped $\lambda\beta$-Calculus

{Refl}	$x = x$
{Sym}	$\dfrac{M = N}{N = M}$
{Trans}	$\dfrac{L = M \qquad M = N}{L = N}$
{Cong}	$\dfrac{M = M' \qquad N = N'}{H \vdash M(N) = M'(N')}$
{ξ}	$\dfrac{M = M'}{\lambda x.\ M = \lambda x.\ M'}$
{β}	$(\lambda x : s.\ M)(N) = [N/x]M$

$P : t$. Of course, distinct well-typed terms may have the same erasure if they differ only in the tags on their bound variables:

$$\mathrm{erase}(\lambda x : \mathbf{o}.\ x) \equiv \lambda x.\ x \equiv \mathrm{erase}(\lambda x : \mathbf{o} \to \mathbf{o}.\ x).$$

6.2 What is a model of the untyped λ-calculus?

It is possible to describe a semantics for the untyped λ-calculus using the simply-typed calculus. Such an interpretation must deal with the concept of self application such as we see in the term $\lambda x.\ x(x)$, so some care must be applied in explaining how an untyped term can be interpreted in a typed setting where this operation is not type-correct. The approach we use, which follows [Meyer, 1982], is to view the application as entailing an *implicit coercion* that converts an application instance of a value into a corresponding function. More precisely, assume we are given constants

$$\Phi : \mathbf{o} \to (\mathbf{o} \to \mathbf{o})$$
$$\Psi : (\mathbf{o} \to \mathbf{o}) \to \mathbf{o}$$

and an equational theory with one equation

$$\Phi \circ \Psi = \lambda x : \mathbf{o} \to \mathbf{o}.\ x. \tag{6.1}$$

Let us call this *theory* U^β. A model of theory U^β is a tuple

$$(\mathcal{A}, A, \Phi, \Psi)$$

where $(\mathcal{A}, A)$ is a type frame and $\Phi \in D^{\mathbf{o}\to(\mathbf{o}\to\mathbf{o})}$ and $\Psi \in D^{(\mathbf{o}\to\mathbf{o})\to\mathbf{o}}$ satisfy 6.1. (To simplify the notation, let us make no distinction between Φ and Ψ as constant symbols in the calculus and their interpretations in the model.) We may use a model of theory U^β to interpret the untyped $\lambda\beta$-calculus in the following way. First, we define a *syntactic translation* that converts an untyped term into a term with type tags by induction as follows:

$$\begin{aligned} &x^* \equiv x \\ &(\lambda x.\ M)^* \equiv \Psi(\lambda x : \mathbf{o}.\ M^*) \\ &(M(N))^* \equiv \Phi(M^*)(N^*) \end{aligned}$$

For example, Y and Y^* are as follows:

$$\begin{aligned} &Y \equiv \lambda f.\ (\lambda x.\ f(xx))(\lambda x.\ f(xx)) \\ &Y^* \equiv \\ &\quad \Psi(\lambda f : \mathbf{o}.\ \Phi(\Psi(\lambda x : \mathbf{o}.\ \Phi(f)(\Phi(x)(x))))(\Psi(\lambda x : \mathbf{o}.\ \Phi(f)(\Phi(x)(x))))). \end{aligned}$$

It is possible to demonstrate the following basic fact about the translation:

Proposition 6.2.1. *Let M be an untyped term. If $H \equiv x_1 : \mathbf{o}, \ldots, x_n : \mathbf{o}$ is a type context that includes all of the free variables of M, then $H \vdash M^* : \mathbf{o}$.*

With this translation, it is possible to assign a meaning to an untyped term M by taking $\mathcal{A}_u[\![M]\!] = \mathcal{A}[\![M^*]\!] \in D^{\mathbf{o}}$. To see that this respects the equational rules given in Table 16, note first the following:

Lemma 6.2.2. $[N^*/x]M^* \equiv ([N/x]M)^*$.

We then prove the β-rule for the untyped calculus by a calculation in the typed one:

$$\begin{aligned} ((\lambda x.\ M)(N))^* &= \Phi((\lambda x.\ M)^*)(N^*) \\ &\equiv \Phi(\Psi(\lambda x : \mathbf{o}.\ M^*))(N^*) \\ &\equiv (\lambda x : \mathbf{o}.\ M^*)(N^*) \\ &= [N^*/x]M^* \\ &\equiv ([N/x]M)^* \end{aligned}$$

so

$$\mathcal{A}_u[\![(\lambda x.\ M)(N)]\!] = \mathcal{A}[\![((\lambda x.\ M)(N))^*]\!] = \mathcal{A}[\![([N/x]M)^*]\!] = \mathcal{A}_u[\![[N/x]M]\!].$$

The other axioms and rules are not difficult.

Now, let *theory* $U^{\beta\eta}$ be theory U^{β} together with the equation

$$\Psi \circ \Phi = \lambda f : \mathbf{o}.\ f, \tag{6.2}$$

which asserts, in effect, that Φ is an isomorphism between the ground type $D^{\mathbf{o}}$ and the functions in $D^{\mathbf{o}\to\mathbf{o}}$. If a model $\mathcal{A}$ of theory U^{β} is also a model of theory $U^{\beta\eta}$, then $\mathcal{A}_u$ satisfies the η-rule as well as the β-rule. To see this, suppose the variable x does not appear free in the term M, then

$$\begin{array}{rcl} (\lambda x.\ M(x))^* & \equiv & \Psi(\lambda x : \mathbf{o}.\ (M(x))^*) \\ & \equiv & \Psi(\lambda x : \mathbf{o}.\ \Phi(M^*)(x)) \\ & = & \Psi(\Phi(M^*)) \\ & = & M^* \end{array}$$

so $\mathcal{A}_u[\![\lambda x.\ M(x)]\!] = \mathcal{A}[\![(\lambda x.\ M(x))^*]\!] = \mathcal{A}[\![M^*]\!] = \mathcal{A}_u[\![M]\!]$. In summary, we have the following result.

Theorem 6.2.3. *If $\mathcal{A}$ is a model of U^{β}, then A_u is a model of the untyped $\lambda\beta$-calculus. If, moreover, $\mathcal{A}$ is a model of $U^{\beta\eta}$, then it is a model of the untyped $\lambda\beta\eta$-calculus.*

6.3 What models of the untyped λ-calculus are there?

Having established that models of theories of U^{β} and $U^{\beta\eta}$ provide models of the untyped $\lambda\beta$ and $\lambda\beta\eta$ calculi respectively, it is tempting to conclude that we have almost completed our quest for models of the untyped calculus. In fact, we have only done the *easy* part. We have not yet shown that any models of theories U^{β} and $U^{\beta\eta}$ actually *exist.* To see why this might be a problem, consider the full type frame $\mathcal{F}_X$ over a set X. If we can find functions

$$\begin{array}{l} \Phi : X \to (X \to X) \\ \Psi : (X \to X) \to X \end{array}$$

(where $X \to X$ is the set of all functions from X to X) that satisfy Equation 6.1, then we have produced the desired model. But Equation 6.1 implies that the function Φ is a *surjection* from X onto the set of functions $f : X \to X$. By Cantor's Theorem, such a surjection exists only if X has exactly one point! This means that the full type frame can only yield a trivial model for the untyped calculus through a choice of Φ and Ψ. The problem lies in the fact that the interpretation of $\mathbf{o} \to \mathbf{o}$ in the full type frame has *too many functions.* To find a model of the untyped calculus, we must therefore look for a type frame that has a more parsimonious interpretation of the higher types.

Techniques for finding models that can satisfy these properties were the primary purpose of the theory of domains and it is beyond the scope of this chapter to discuss how this can be done in any detail. For the sake of concreteness, however, let us build one such domain explicitly. Given a set X, let $\mathcal{P}_f(X)$ be the set of all finite subsets of X. Define an operation G by

$$G(X) = \{(u, x) \mid u \in \mathcal{P}_f(X) \text{ and } x \in X\}.$$

Starting with any set X, let $X_0 = X$ and $X_{n+1} = G(X_n)$. Take D_X to be the set $\mathcal{P}(\bigcup_{n \in \omega} X_n)$ of all subsets of the union of the X_n's. Ordered by set inclusion, this is an algebraic lattice whose compact elements are the finite sets. To see how it can be viewed as a model of the untyped λ-calculus, consider an element $(u, x) \in G(X_n)$. This pair can be viewed as a piece of a function f, which has x in its output whenever u is a subset of its input. Suppose that $d \in D_X$. Following this intuition, we define a function $\Phi(d) : D_X \to D_X$ by taking

$$\Phi(d)(e) = \{x \mid u \subseteq e \text{ for some } (u, x) \in d\} \tag{6.3}$$

for each $e \in D_X$. In other words, if we are to view d as inducing a function taking e as its argument, the result of applying d to e is the set of those elements x such that there is a 'piece' (element) (u, x) of d where u is a subset of the input e. Also, given a continuous function $f : D_X \to D_X$, define $\Psi(f) \in D_X$ by

$$\Psi(f) = \{(u, x) \mid x \in f(u)\}. \tag{6.4}$$

This says that f is to be represented by recording the pairs (u, x) such that x will be part of the result of applying f to an element that contains u. It is not difficult to check that Φ and Ψ are continuous. Suppose that $f : D_X \to D_X$ is continuous. Then

$$\begin{array}{rcl} \Phi(\Psi(f))(d) & = & \{x \mid (u, x) \in \Psi(f) \text{ for some } u \subseteq d\} \\ & = & \{x \mid x \in f(u) \text{ for some } u \subseteq d\} \\ & = & \bigcup\{f(u) \mid u \subseteq d\} \\ & = & f(d) \end{array}$$

where the last step follows from the fact that a continuous function on an algebraic cpo is determined by its action on compact elements.

A model of the simply-typed λ-calculus is defined by taking D_X as the interpretation of the ground type and interpreting higher types using the (pointwise ordered) continuous function spaces. It then follows from the calculation above that the continuous type frame generated by D_X, together with the continuous functions Φ and Ψ, is a model of the untyped $\lambda\beta$-calculus. It is obviously non-trivial if X has at least two distinct elements.

Could D_X also be a model of the untyped $\lambda\beta\eta$-calculus? Suppose $d \in D_X$, and let us attempt to calculate equation 6.2:

$$\begin{aligned}\Psi(\Phi(d)) &= \{(u,x) \mid x \in \Phi(d)(u)\} \\ &= \{(u,x) \mid v \subseteq u \text{ for some } (v,x) \in d\} \\ &\supseteq d.\end{aligned}$$

But since d could be an arbitrary subset of $\bigcup_n X_n$, it is clear that this superset relation may fail to be an *equality.* So we have not yet demonstrated a model for the $\lambda\beta\eta$-calculus! To do this using cpo's and continuous functions, what we need is a cpo D such that $D \cong [D \to D]$. Here is where domain theory would be helpful in finding spaces with the needed properties. Let us assum that we know how to find non-trivial domains to satisfy this and other isomorphisms; for details on how such domains can be constructed, see [Abramsky and Jung, 1994].

6.4 Inclusive subsets as types.

A domain used as a universe for interpreting types is generally called a *universal domain.* It is not desirable to use arbitrary subsets of a universal domain to denote types because certain properties are needed to establish invariants. Let us now consider one of the most widely used conditions.

Definition 6.4.1. A subset of a cpo is *inclusive* if it is downward closed and closed under least upper bound's of ω-chains.

The use of the term 'inclusive' for this notion is slightly non-standard from a historical perspective. The condition above was introduced in [Milner, 1978], and it is common for the term 'ideal' to be used instead [MacQueen *et al.*, 1986]. This clashes with another common usage of 'ideal', so the alternate term 'inclusive' is used here.

Based on a given universal domain, it is possible to use inclusive subsets to model simple types. Let X be any domain and suppose we are given a solution to the domain equation

$$D \cong X \oplus [D \to D]. \tag{6.5}$$

In this equation, the operator $\oplus$ is the *coalesced sum*, which takes the disjoint union of its arguments and then identifies their respective least elements. This is a model of the untyped λ-calculus: let $\Psi : [D \to D] \to D$ be the injection of the cpo of continuous functions from D to D into the right component of D (note that the function $x \mapsto \bot_D$ is being set to $\bot_D$) and let $\Phi : D \to [D \to D]$ be given by

$$\Phi(y) = \begin{cases} f & \text{if } y = \Psi(f) \text{ for a continuous } f : D \to D \\ \bot_{[D \to D]} & \text{if } y \in X. \end{cases}$$

It is easy to see that $\Phi \circ \Psi = \mathbf{id}$. To associate inclusive subsets on D with types of the simply-typed λ-calculus, define

$$\begin{aligned}&[\![\mathbf{o}]\!] = X\\ &[\![s \to t]\!] = \{\Psi(f) \mid f \in [D \to D] \text{ and } f(x) \in [\![t]\!] \text{ whenever } x \in [\![s]\!]\}.\end{aligned}$$

Lemma 6.4.2. *For each type t, the subset $[\![t]\!] \subseteq D$ is an inclusive subset.*

Given a type assignment H, an H-environment ρ is defined to be a function from variables into D such that $\rho(x) \in [\![H(x)]\!]$ for each $x \in H$. If $H \vdash M : t$, then the interpretation of M is given by induction on the structure of M relative to an H-environment ρ as follows:

$$\begin{aligned}&[\![x]\!]\rho = \rho(x)\\ &[\![L(N)]\!]\rho = \Phi([\![L]\!]\rho)([\![N]\!]\rho)\\ &[\![\lambda x : s.\ M']\!]\rho = \Psi(d \mapsto [\![M']\!]\rho[x \mapsto d])\end{aligned}$$

The key result relating this interpretation to the interpretations of types is the following:

Theorem 6.4.3. *If $H \vdash M : t$ and ρ is an H-environment, then $[\![M]\!]\rho \in [\![t]\!]$.*

There is a problem with interpreting types in this way, however: the equational rules are not all satisfied. To see why this is the case, suppose that the domain X in Equation 6.5 has at least one element other than $\bot_X$. Define two terms

$$\begin{aligned}&M \equiv \lambda y : s \to t.\ \lambda x : s.\ y(x)\\ &N \equiv \lambda y : s \to t.\ y.\end{aligned}$$

The meaning of M can be calculated as

$$\begin{aligned}[\![M]\!]\emptyset &= \Psi(e \mapsto [\![\lambda x : s.\ y(x)]\!][y \mapsto e])\\ &= \Psi(e \mapsto \Psi(d \mapsto [\![y(x)]\!][y, x \mapsto e, d]))\\ &= \Psi(e \mapsto \Psi(d \mapsto \Phi([\![y]\!][y, x \mapsto e, d])([\![x]\!][y, x \mapsto e, d])))\\ &= \Psi(e \mapsto \Psi(d \mapsto \Phi(e)(d)))\end{aligned}$$

and the value of N by

$$[\![N]\!]\emptyset = \Psi(d \mapsto [\![y]\!][y \mapsto d]) = \Psi(d \mapsto d).$$

Suppose $\bot \neq a \in [\![\mathbf{o}]\!]$. Then $[\![M]\!] \neq [\![N]\!]$ because $\Phi[\![M]\!]\emptyset(a) \neq \Phi[\![N]\!]\emptyset(a)$. To see why, first calculate

$$\Phi([\![M]\!]\emptyset)(a) \quad = \quad \Phi(\Psi(e \mapsto \Psi(d \mapsto \Phi(e)(d))))(a)$$

$$
\begin{aligned}
&= \Psi(d \mapsto \Phi(a)(d)) \\
&= \Psi(d \mapsto (e \mapsto \bot)(d)) \\
&= \Psi(d \mapsto \bot) \\
&= \bot
\end{aligned}
$$

whereas

$$\Phi([\![N]\!]\emptyset)(a) = \Phi(\Psi(d \mapsto d))(a) = a \neq \bot.$$

But these two terms are provably equal in the equational theory. Let $r \equiv s \to t$, then by projection

$$\vdash (y : r \rhd y = y : r)$$

so, by the η-rule,

$$\vdash (y : r \rhd \lambda x : s.\ y(x) = y : r)$$

Hence, by the ξ-rule,

$$\vdash (\rhd\ \lambda y : r.\ \lambda x : s.\ y(x) = \lambda y : r.\ y : r \to r).$$

This is the equation $M = N$ that is not satisfied by the model. Where does the problem lie here? It is not the η-rule as one might originally suspect: the η-rule is satisfied by the interpretation. The problem is the soundness of the ξ-rule: the terms M and N denote functions, but if they are applied (in the model) to elements of the 'wrong type', then the values may differ.

A construction that solves this problem will be given below, but let us consider how this model is useful even without satisfying the full equational theory for the λ-calculus. The idea presented in [Milner, 1978] is to give a semantic proof of a result such as Theorem 5.1.1. Milner's proof was for the calculus ML_0, but the idea can be illustrated adequately with PCF. Let us look at the analog of that theorem for PCF with type errors (that is, PCF with a new expression **tyerr** that does not have a type and expanding the operational rules in Table 8 to include rules for type errors as given in Table 11).

To give a fixed-point model of the calculus, we use a domain U for which there is an isomorphism

$$U \cong \mathbb{T} \oplus \mathbb{N}_\bot \oplus [U \to U]_\bot \oplus \mathbb{O}. \tag{6.6}$$

In this equation, the operation $D \mapsto D_\bot$ is the *lift*, which adds a new bottom element to the domain D. There are obvious maps $\mathbf{up} : D \to D_\bot$ and $\mathbf{down} : D_\bot \to D$ relating this domain to D. The space $\mathbb{O}$ is the two-point lattice; let us denote its non-bottom element by **tyerr** to save some notation, since it will be used as the meaning of the term **tyerr**. Let us write $d : D$ for the injection of element d into the D component of the sum. For example, $n : \mathbb{N}_\perp$ is the injection of $n \in \mathbb{N}_\perp$ into the second component of U.

Now, as usual, the semantics of a term M such that $H \vdash M : t$ is relative to an H-environment ρ.

- $\mathcal{I}[\![x]\!]\rho = \rho(x)$.
- $\mathcal{I}[\![\lambda x : t.\ M']\!]\rho = \mathbf{up}(f) : [U \to U]_\perp$ where

$$f(d) = \begin{cases} \mathbf{tyerr} & \text{if } d = \mathbf{tyerr} \\ \mathcal{I}[\![M']\!]\rho[x \mapsto d] & \text{otherwise.} \end{cases}$$

- $\mathcal{I}[\![L(N)]\!]\rho = \begin{cases} \mathbf{down}(f)(\mathcal{I}[\![N]\!]\rho) & \text{if } \mathcal{I}[\![L]\!]\rho = f : [U \to U]_\perp \\ \mathbf{tyerr} & \text{otherwise.} \end{cases}$
- $\mathcal{I}[\![\mu x : t.\ M']\!]\rho = \mathbf{fix}(d \mapsto \mathcal{I}[\![M']\!]\rho[x \mapsto d])$.
- $\mathcal{I}[\![\mathbf{tyerr}]\!]\rho = \mathbf{tyerr}$.

The remaining constructs of PCF have a straight-forward interpretation that makes the following true:

Lemma 6.4.4.

1. *For each type t, $\mathcal{I}[\![t]\!]$ is an inclusive subset.*
2. *If $H \vdash M : t$ and ρ is an H-environment, then $\mathcal{I}[\![M]\!]\rho \in \mathcal{I}[\![t]\!]$*
3. *If $M \Downarrow V$, then $\mathcal{I}[\![M]\!] = \mathcal{I}[\![V]\!]$.*

The point of the Lemma is the following:

Theorem 6.4.5. *If $M : t$ and $M \Downarrow V$, then V is not* **tyerr**.

To see this, note that $\mathcal{I}[\![M]\!] = \mathcal{I}[\![V]\!]$ and $\mathcal{I}[\![M]\!]\emptyset \in \mathcal{I}[\![t]\!]$. Since $\mathbf{tyerr} \notin \mathcal{I}[\![t]\!]$ it follows that $\mathcal{I}[\![V]\!]\emptyset \neq \mathbf{tyerr}$ so it cannot be the case that V is **tyerr**.

6.5 Subtyping as subset inclusion.

Let us return now to the topic of a *subtype* as discussed in Section 2.4. In that discussion, the idea that a sub*type* is a sub*set* was used as an intuition. Using inclusive predicates, it is possible to make this intuition into a formal model. To illustrate this, let us consider a pair of extensions of PCF. The type system considered here is based on ideas in [Reynolds, 1980] and [Cardelli, 1988]; the semantics is basically the one in [Cardelli, 1988].

Table 17. Typing Rules for Records and Variants

[RecIntro]	$$\frac{H \vdash M_1 : t_1 \quad \cdots \quad H \vdash M_n : t_n}{H \vdash \{l_1 = M_1, \ldots, l_n = M_n\} : \{l_1 : t_1, \ldots, l_n : t_n\}}$$
[RecElim]	$$\frac{H \vdash M : \{l_1 : t_1, \ldots, l_n : t_n\}}{H \vdash M.l_i : t_i}$$
[VarIntro]	$$\frac{H \vdash M : t}{H \vdash [l = M, l_1 : t_1, \ldots, l_n : t_n] : [l : t, l_1 : t_1, \ldots, l_n : t_n]}$$
[VarElim]	$$\frac{\begin{array}{l} H \vdash M : [l_1 : t_1, \ldots, l_n : t_n] \\ H \vdash M_1 : t_1 \to t \quad \cdots \quad H \vdash M_n : t_n \to t \end{array}}{H \vdash \mathbf{case}\ M\ \mathbf{of}\ l_1 \Rightarrow M_1 \cdots l_n \Rightarrow M_n : t}$$

First we extend PCF to a language that includes records and variants; this extension is called *PCF+* or 'PCF plus records and variants'. To define its terms, we require a primitive syntax class of *labels* $l \in$ Label. Here is its grammar:

$$\begin{array}{rcl}
x & \in & \text{Variable} \\
l & \in & \text{Label} \\
t & ::= & \mathbf{num} \mid \mathbf{bool} \mid t \to t \mid \{l : t, \ldots, l : t\} \mid [l : t, \ldots, l : t] \\
M & ::= & \mathbf{0} \mid \mathbf{true} \mid \mathbf{false} \mid \\
 & & \mathbf{succ}(M) \mid \mathbf{pred}(M) \mid \mathbf{zero?}(M) \mid \mathbf{if}\ M\ \mathbf{then}\ M\ \mathbf{else}\ M \mid \\
 & & x \mid \lambda x : t.\ M \mid MM \mid \mu x : t.\ M \mid \\
 & & \{l = M, \ldots, l = M\} \mid M.l \mid \\
 & & [l = M, l : t, \ldots, l : t] \mid \mathbf{case}\ M\ \mathbf{of}\ l \Rightarrow M, \ldots, l \Rightarrow M
\end{array}$$

where ellipsis (the notation with three dots) is used to indicate finite lists of pairs in records and variants. The types and terms of PCF+ are of expressions generated by this grammar for which there are no repeated labels in the lists of label/type and label/term pairs. The terms of PCF+ are taken modulo α-conversion (renaming of bound variables) and the order in which the fields in records and variants are written. A similar equivalence is assumed for record and variant type expressions.

The typing rules for PCF+ are those for PCF in Tables 4 and 6, together with rules for records and variants given in Table 17. They are quite similar to the rules for products and sums. As with the sum, it is essential to label the variations to ensure that a variant has a unique type. In general we have the following:

Table 18. Rules for Subtyping

$$\mathbf{num} \leq \mathbf{num} \qquad \frac{s' \leq s \qquad t \leq t'}{s \to t \leq s' \to t'}$$
$$\mathbf{bool} \leq \mathbf{bool}$$

$$\frac{s_1 \leq t_1 \quad \cdots \quad s_n \leq t_n}{\{l_1 : s_1, \ldots, l_n : s_n, \ldots, l_m : s_m\} \leq \{l_1 : t_1, \ldots, l_n : t_n\}}$$

$$\frac{s_1 \leq t_1 \quad \cdots \quad s_n \leq t_n}{[l_1 : s_1, \ldots, l_n : s_n] \leq [l_1 : t_1, \ldots, l_n : t_n, \ldots, l_m : t_m]}$$

Theorem 6.5.1. *If $H \vdash M : s$ and $H \vdash M : t$, then $s \equiv t$.*

This will not be true of the calculus PCF++ we now consider. PCF++ is the extension of PCF+ in which we allow the use of a subtyping relation between types. The binary relation $s \leq t$ of subtyping between type expressions s and t is defined by the rules in Table 18. It is possible to show that $\leq$ is a poset on type expressions. The calculus PCF++ is the same as PCF+ but with the typing rules of PCF+ extended by the addition of the *subsumption rule.* Since a type can be derived for a term using subsumption that could not be derived without it, it will be essential to distinguish between typing judgements for PCF++ and those of PCF+. Let us write $\vdash_{\mathrm{sub}}$ for the least relation that contains the relation $\vdash$ of PCF+ and satisfies

$$[\text{Subsump}] \qquad \frac{H \vdash_{\mathrm{sub}} M : s \qquad s \leq t}{H \vdash_{\mathrm{sub}} M : t}.$$

The operational semantics of PCF+ and PCF++ is similar to that of PCF but the language is evaluated using call-by-value rather than call-by-name. The rules in Table 8 are used for PCF+ and PCF++ except for the rule:

$$\frac{M \Downarrow \lambda x : s.\ M' \qquad [N/x]M' \Downarrow V}{M(N) \Downarrow V}$$

for evaluation of applications, which is replaced by the rule:

$$\frac{M \Downarrow \lambda x : s.\ L \qquad N \Downarrow U \qquad [U/x]L \Downarrow V}{M(N) \Downarrow V}$$

The evaluation of records and variants is given by the rules in Table 19.

A model of PCF++ can be given by extending the inclusive subsets interpretation for PCF. To this end we need a domain similar to the one in Equation 6.6. Let us define

$$U \cong \mathbb{T} \oplus \mathbb{N}_\perp \oplus [U \to U]_\perp \oplus [\text{Label} \to U]_\perp \oplus (\text{Label} \times U)_\perp \oplus \mathbb{O}. \qquad (6.7)$$

Table 19. Rules for Call-by-Value Evaluation of Records and Variants

$$\frac{M_1 \Downarrow V_1 \quad \cdots \quad M_n \Downarrow V_n}{\{l_1 = M_1, \ldots, l_n = M_n\} \Downarrow \{l_1 = V_1, \ldots, l_n = V_n\}}$$

$$\frac{M \Downarrow \{l_1 = V_1, \ldots, l_n = V_n\}}{M.l_i \Downarrow V_i}$$

$$\frac{M \Downarrow V}{[l = M, \ldots] \Downarrow [l = V, \ldots]}$$

$$\frac{M \Downarrow [l_i = U, \ldots] \qquad f_i(U) \Downarrow V}{\mathbf{case}\ M\ \mathbf{of}\ l_1 \Rightarrow f_1, \ldots, l_i \Rightarrow f_i, \ldots, l_n \Rightarrow f_n \Downarrow V}$$

As before, let us write $d : D$ for the injection of element d into the D component of the sum. For example, $\mathbf{up}(f) : [U \to U]_\perp$ is the injection of a continuous function $f : U \to U$ into the third component of U. The semantics of types is defined as follows:

- $\mathcal{I}[\![\ \{l_1 : t_1, \ldots, l_n : t_n\}\ ]\!] = \{r : [\mathrm{Label} \to U]_\perp \mid \mathbf{down}(r)(l_i) \in \mathcal{I}[\![t_i]\!]$ for each $i = 1, \ldots, n\}$
- $\mathcal{I}[\![\ [l_1 : t_1, \ldots, l_n : t_n]\ ]\!] = \{e : (\mathrm{Label} \times U)_\perp \mid e = \perp$, or $\mathbf{down}(e) = (l_i, d)$ and $d \in \mathcal{I}[\![t_i]\!]\}$
- $\mathcal{I}[\![s \to t]\!] = \{f : [U \to U]_\perp \mid \mathbf{down}(f)(d) \in \mathcal{I}[\![t]\!]$ for each $d \in \mathcal{I}[\![s]\!]\}$

The semantics of a term M such that $H \vdash_{\mathrm{sub}} M : t$ is relative to an H-environment ρ.

- $\mathcal{I}[\![\ \{l_1 = M_1, \ldots, l_n = M_n\}\ ]\!]\rho = \mathbf{up}(r) : [\mathrm{Label} \to U]_\perp$ where

$$r(l) = \begin{cases} \mathcal{I}[\![M_i]\!]\rho & \text{if } l = l_i \\ \mathbf{tyerr} & \text{otherwise} \end{cases}$$

- $\mathcal{I}[\![M.l]\!]\rho = \begin{cases} \mathbf{down}(f)(l) & \text{if } \mathcal{I}[\![M]\!]\rho = f : [\mathrm{Label} \to U]_\perp \\ \mathbf{tyerr} & \text{otherwise} \end{cases}$
- $\mathcal{I}[\![\ [l = M, l_1 : t_1, \ldots, l_n : t_n]\ ]\!]\rho = \mathbf{up}(l, \mathcal{I}[\![M]\!]\rho) : (\mathrm{Label} \times U)_\perp$
- $\mathcal{I}[\![\mathbf{case}\ M\ \mathbf{of}\ l_1 \Rightarrow M_1 \cdots l_n \Rightarrow M_n]\!]\rho = d$ where
    - $d = \mathbf{down}(f_i)(e)$ if $\mathcal{I}[\![M]\!]\rho = \mathbf{up}(l_i, e) : (\mathrm{Label} \times U)_\perp$ and $\mathcal{I}[\![M_i]\!]\rho = f_i : [U \to U]_\perp$

* $d = \bot$ if $\mathcal{I}[\![M]\!]\rho = \bot$
* $d =$ **tyerr** otherwise

It is not difficult to check the following property of the interpretation:

Lemma 6.5.2. *For each type t the subset $\mathcal{I}[\![t]\!]\rho$ inclusive.*

Given a suitable choice of the definitions for arithmetic expressions, it is possible to arrange it to be the case that **tyerr** $\notin \mathcal{I}[\![t]\!]$ for each of the types t of PCF+. The interpretation also allows us the intuitive liberty of thinking of $s \leq t$ as meaning that the meaning of s is a *subset* of the meaning of t:

Theorem 6.5.3. *If $s \leq t$, then $\mathcal{I}[\![s]\!] \subseteq \mathcal{I}[\![t]\!]$.*

The converse of the theorem also holds, if we assume that the solution to Equation 6.7 is an algebraic cpo with a countable basis. Finally, we have the following:

Theorem 6.5.4.

1. *If $H \vdash M : t$ and ρ is an H-environment, then $\mathcal{I}[\![H \rhd M : t]\!]\rho \in \mathcal{I}[\![t]\!]$.*
2. *If $M \Downarrow V$, then $\mathcal{I}[\![M]\!] = \mathcal{I}[\![V]\!]$.*
3. *If $M : t$ and $M \Downarrow V$, then V is not* **tyerr**.

7 Types as Partial Equivalence Relations

Let us resume the discussion of parametric polymorphism begun earlier by considering some of the type systems used to express this notion. Our goal is to study the distinction between *predicative* and *impredicative* definitions of types and show how the latter can be modeled by interpreting types as equivalence relations on subsets of a universal domain. The system ML_0 is an example of a predicative system; we begin by demonstrating a set-theoretic model of this system and considering an alternative presentation of its typing rules. This system is then generalized to the best-known impredicative system, the Girard-Reynolds polymorphic λ-calculus. Modeling the impredicativity of the Girard-Reynolds system demands more subtlety; it is shown how this can be done by interpreting types as equivalence relations on subsets of a universal domain.

7.1 Sets as a model of ML_0 types.

Let us go back now and consider the semantics of ML_0. The goal is to provide a model for polymorphic types analogous to the full type frame for simple types. Recall the syntax of types, type schemes, and terms for the language:

$$\begin{array}{lcl} x & \in & \mathrm{TermVariable} \\ a & \in & \mathrm{TypeVariable} \\ t & ::= & a \mid t \to t \\ T & ::= & t \mid \Pi a.\, T \\ M & ::= & x \mid \lambda x.\, M \mid MM \mid \mathbf{let}\ x = M\ \mathbf{in}\ M \end{array}$$

It will be necessary to have two forms of environment to model the language. Since types may contain variables, we will need the notion of a *type-value environment,* ι which is a function that maps types to semantic domains. An *H, ι-environment* is a mapping ρ that assigns to each $x \in H$ an element $\rho(x) \in [\![H(x)]\!]\iota$. For ML_0 we use sets drawn from a collection obtained by constructing the full type frame.

Let X_0 be any non-empty set; it will serve as the analog of the interpretation of the ground type. For sets S, T, define T^S to be the set of functions from S to T. Define $D_0 = \{X_0\}$ and

$$D_{n+1} = \{Y^X \mid X, Y \in D_n\} \cup D_n.$$

The *universe* of our interpretation is the set $U = \bigcup_{n \in \omega} D_n$. To model types of ML_0, we must interpret type schemes as well as types. Given an operator F such that $F(X)$ is a set for each $X \in U$, define the *dependent product determined by* F to be the set $\Pi_{X \in U} F(X)$ that consists of functions π such that $\pi(X) \in F(X)$ for each $X \in U$. To be more precise about the nature of such maps π, they can be taken as functions with domain U and range $\bigcup_{X \in U} F(X)$ satisfying the given constraint that $\pi(X) \in F(X)$. The interpretation of types and type schemes can now be given as follows:

- $[\![a]\!]\iota = \iota(a)$
- $[\![s \to t]\!]\iota = ([\![t]\!]\iota)^{[\![s]\!]\iota}$
- $[\![\Pi a.\, T]\!]\iota = \Pi_{X \in U} [\![T]\!]\iota[a \mapsto X]$

It is easy to see that $[\![t]\!]\iota$ is an element of U for each type t since the universe U is closed under exponentiation. Note, however, that $[\![\Pi a.\, T]\!]\iota$ need not be an element of U despite the fact that type variables a are mapped to elements of U. Implicitly we are therefore working with two universes. The first of these, U, is used for interpreting types, while the second contains sets that can be the interpretations of type schemes. To be more precise, let $U = V_0$ and, for each $n \in \omega$, define $V_{n+1} = (V_n)^U \cup V_n$. Then the meaning $[\![\Pi a.\, T]\!]\rho$ of a type scheme is an element of a second universe $V = \bigcup_{n \in \omega} V_n$.

The interpretation of the terms of ML_0 is more subtle than that of types. Let me write out the equations for the semantics in full and then consider whether they describe a well-defined function.

- Suppose $H \Vdash x : t$ and $t \leq H(x) \equiv \Pi a_1. \ldots \Pi a_n.\, s$. Let σ be a substitution such that $\sigma(s) \equiv t$. Letting $X_i = [\![\sigma(a_i)]\!]\iota$ for each i, define $[\![H \rhd x : t]\!]\iota\rho = \rho(x)(X_1)\cdots(X_n)$.
- $[\![H \rhd \lambda x.\, M : s \to t]\!]\iota\rho$ is the function from $[\![s]\!]\iota$ to $[\![t]\!]\iota$ defined by $d \mapsto [\![H,\ x : s \rhd M : t]\!]\iota\rho[x \mapsto d]$.
- $[\![H \rhd M(N) : t]\!]\iota\rho = ([\![H \rhd M : s \to t]\!]\iota\rho)([\![H \rhd N : s]\!]\iota\rho)$.
- Suppose $\mathrm{close}(H;\ s) = \Pi a_1. \ldots \Pi a_n.\, s$ and $H \Vdash M : s$. Define $\pi \in [\![\mathrm{close}(H;\ s)]\!]\iota$ by

  $$\pi(X_1)\cdots(X_n) = [\![H \rhd M : s]\!](\iota[a_1, \ldots, a_n \mapsto X_1, \ldots, X_n])\rho.$$

  Then $[\![H \rhd \mathbf{let}\ x = M\ \mathbf{in}\ N : t]\!]\iota\rho = [\![H, x : \mathrm{close}(H;\ s) \rhd N : t]\!]\iota\rho[x \mapsto \pi]$.

The primary question about the sense of this definition concerns whether the type-value environment $\iota' = \iota[a_1, \ldots, a_n \mapsto X_1, \ldots, X_n]$ in the semantic equation for the **let** construct is compatible with the environment ρ; that is, whether ρ is an H, ι'-environment. This question is resolved by recalling that none of the type variables a_i is in $\mathrm{Ftv}(H)$ and by noting the following:

Lemma 7.1.1. *If ρ is an H, ι-environment and $a \notin \mathrm{Ftv}(H)$, then it is also an $H, \iota[a \mapsto X]$-environment.*

which follows from the fact that $[\![t]\!]\iota[a \mapsto X] = [\![t]\!]\iota$ if a is not free in t.

7.2 Another typing system for ML_0.

In light of the semantics we just gave, the type system we have been using for ML_0 appears to be slightly indirect in some ways. In the rule [Let], for instance, the meaning of the term M in **let** $x = M$ **in** N is calculated, and then a 'parameterized' version of its meaning is bound to x in the environment. Similarly, the meaning of a variable is drawn from the environment and then instantiated to the type assigned to x. Permitting judgements of the form $H \vdash M : T$, where T is a type scheme, together with rules for generalization and instantiation might lead a more elementary system. It

Table 20. Typing Rules for ML_0 with Π Introduction and Elimination

[Proj]	$\dfrac{x : T \in H}{H \vdash x : T}$
$[\mathrm{Abs}]^{-}$	$\dfrac{H,\ x : s \vdash M : t}{H \vdash \lambda x.\ M : s \to t}$
[Appl]	$\dfrac{H \vdash M : s \to t \qquad H \vdash N : s}{H \vdash M(N) : t}$
[Let]	$\dfrac{H \vdash M : T \qquad H, x : T \vdash N : t}{H \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : t}$
$[\Pi\text{-Intro}]^{-}$	$\dfrac{H \vdash M : T \qquad a \notin \mathrm{Ftv}(H)}{H \vdash M : \Pi a.\ T}$
$[\Pi\text{-Elim}]^{-}$	$\dfrac{H \vdash M : \Pi a.\ T}{H \vdash M : [t/a]T}$

is indeed possible to reformulate the typing system for ML_0 in this way. The rules for deriving such judgements appear in Table 20. The rules for abstraction and application remain unchanged, but the rules for projections and **let** constructs now reflect the more general form of judgement in this system. In the new system, the projection rule looks more or less the way it does in most of the systems we have considered rather than having the somewhat different form it has in Table 13. The 'close' operator is no longer used in the rule for **let** since the term M in the hypothesis may be given a type scheme rather than a type that must be generalized on variables not in H. But the real difference in the two systems lies in the presence of rules for introduction and elimination of Π bindings. One particular difference made by the addition of these rules is the fact that the derivation of a judgement $H \vdash M : T$ is not uniquely determined by H, M, T. There will, in fact, be many (superficially) distinct proofs of any such judgement obtained by alternating the application of rules $[\Pi\text{-Intro}]^{-}$ and $[\Pi\text{-Elim}]^{-}$. Nevertheless, it is possible to show that two systems are essentially the same on judgements of types.

Proposition 7.2.1. *Let H be a type assignment, M a term, and t a type.*

1. *If $H \Vdash M : t$, then $H \vdash M : t$.*
2. *If $H \vdash M : T$ and $t \leq T$, then $H \Vdash M : t$.*

7.3 The polymorphic λ-calculus.

In many of the calculi we have considered, type annotations in terms were used to force each typeable term to have a unique type. The rules tagged with a superscript minus sign in the typing system for ML_0 described in Table 20 cause this property to fail. To recover it, we might place type tags on the λ-bound variables, but the rules $[\Pi\text{-Intro}]^-$, $[\Pi\text{-Elim}]^-$ would still pose a problem. In effect, terms must contain some indication about the generalization and instantiation of type variables if their types are to be uniquely determined. Let us now consider an important generalization of ML_0 that has explicit abstraction and application for type variables in terms. The system is sometimes called the *Girard-Reynolds* polymorphic λ-calculus, since the system was discovered independently by Girard [Girard, 1972] (who was working on a proof-theoretic problem) and by Reynolds [Reynolds, 1974] (who was interested in programming language design). With the possible exception of 'ML polymorphism', it is the best-known polymorphic type system, so it is most often simply called the *polymorphic λ-calculus.* The reader is refered to the survey paper [Scedrov, 1990] for a fuller discussion of polymorphic λ-calculus, including references and historical background. The terms of the language are given as follows:

$$\begin{array}{rcl} x & \in & \mathrm{TermVariable} \\ a & \in & \mathrm{TypeVariable} \\ t & ::= & t \to t \mid a \mid \Pi a.\ t \\ M & ::= & x \mid \lambda x : t.\ M \mid M(M) \mid \Lambda a.\ M \mid M\{t\} \end{array}$$

A term of the form $\Lambda a.\ M$ is called a *type abstraction,* and one of the form $M\{t\}$ is called a *type application.* Types of the form $\Pi a.\ t$ are called Π-*types,* and the type variable a is bound in $\Pi a.\ t$ by the Π-quantification. The following clauses define the free type variables for types and terms:

- $\mathrm{Ftv}(a) = a$
- $\mathrm{Ftv}(s \to t) = \mathrm{Ftv}(s) \cup \mathrm{Ftv}(t)$
- $\mathrm{Ftv}(\Pi a.\ t) = \mathrm{Ftv}(t) - \{a\}$
- $\mathrm{Ftv}(x) = \emptyset$
- $\mathrm{Ftv}(\lambda x : t.\ M) = \mathrm{Ftv}(t) \cup \mathrm{Ftv}(M)$
- $\mathrm{Ftv}(M(N)) = \mathrm{Ftv}(M) \cup \mathrm{Ftv}(N)$
- $\mathrm{Ftv}(\Lambda a.\ M) = \mathrm{Ftv}(M) - \{a\}$
- $\mathrm{Ftv}(M\{t\}) = \mathrm{Ftv}(M) \cup \mathrm{Ftv}(t)$

For an assignment H, the set of free type variables $\mathrm{Ftv}(H)$ in H is the union of the free type variables in $H(x)$ for each $x \in H$. Substitution

Table 21. Typing Rules for the Polymorphic λ-Calculus

[Π-Intro]	$\dfrac{H \vdash M : s \qquad a \notin \mathrm{Ftv}(H)}{H \vdash \Lambda a.\ M : \Pi a.\ s}$
[Π-Elim]	$\dfrac{H \vdash M : \Pi a.\ s}{H \vdash M\{t\} : [t/a]s}$

for both types and terms must respect bindings in the sense that no free variable of the term being substituted can be captured by a binding in the term into which the substitution is made.

The types for terms of the polymorphic λ-calculus may be built using type variables. For example, $\lambda y : a.\ \lambda x : a \to b.\ x(y)$ is a well-typed term with type $a \to (a \to b) \to b$. However, unlike the ML_0 systems, the polymorphic λ-calculus allows type variables to be explicitly abstracted in terms. For example, the term

$$M \equiv \Lambda a.\ \Lambda b.\ \lambda y : a.\ \lambda x : a \to b.\ x(y)$$

has the type $\Pi a.\ \Pi b.\ a \to (a \to b) \to b$. It is possible to instantiate the abstracted type variables through a form of application. Given types s and t, for example, $M\{s\}\{t\}$ is equivalent to the term $\lambda y : s.\ \lambda x : s \to t.\ x(y)$. This latter term has the type $s \to (s \to t) \to t$.

The precise typing rules for the polymorphic λ-calculus are those in Table 4 together with the two rules that appear in Table 21. Of course, the rules in Table 4 must be understood as applying to all of the terms of the polymorphic calculus as given in the grammar for the language (and not just to the terms of the simply-typed calculus). As with earlier calculi, the type tags on the bound variables ensure the following:

Lemma 7.3.1. *For any type assignment H, term M, and type expressions s, t, if $H \vdash M : s$ and $H \vdash M : t$ then $s \equiv t$.*

The virtue of the polymorphic λ-calculus is that it can be used to express general algorithms in a clear way. For example, let us return to the problem illustrated earlier by the program

```
(define applyto
  (lambda (f) (cons (f 3) (f "hi"))))
```

The function `applyto` takes a function as an argument and applies it to each of the components of a given pair, returning a pair as a result. Here

Table 22. Equational Rules for the Polymorphic λ-Calculus

{TypeCong}	$\dfrac{\vdash (H \rhd M = N : \Pi a.\ t)}{\vdash (H \rhd M\{s\} = N\{s\} : [s/a]t)}$
{Type ξ}	$\dfrac{\vdash (H \rhd M = N : t)}{\vdash (H \rhd \Lambda a.\ M = \Lambda a.\ N : \Pi a.\ t)}$
{Type β}	$\dfrac{H \vdash M : t}{\vdash (H \rhd (\Lambda a.\ M)\{s\} = [s/a]M : [s/a]t)}$
{Type η}	$\dfrac{H \vdash M : \Pi a.\ t}{\vdash (H \rhd \Lambda a.\ M\{a\} = M : \Pi a.\ t)}$

Restrictions:

- In {Type ξ}, there is no free occurrence of a in the type of a variable in H.
- In {Type η}, the type variable a does not appear free in H or M.

is a similar program written in the polymorphic λ-calculus extended with pairs:

$$\Lambda a.\ \lambda f : (\Pi b.\ b \rightarrow a).\ (f\{\text{int}\}(M), f\{\text{string}\}(N))$$

The types must be explicitly instantiated as part of the application, but the program is more general than any that can be written in ML_0. More convincing programming examples can be given, but this shows that the phenomenon arises quite naturally.

The equational rules for the pure polymorphic λ-calculus are those in Table 5 together with the rules that appear in Table 22 modulo the theory T that appears on the left-hand sides of the turnstiles.[1] The new rules {TypeCong} and {Type ξ} assert that type application and type abstraction are congruences. The most fundamental new rules are the type-level analogs {Type β} and {Type η} of the β and η rules.

7.4 Sets as a model of polymorphic types?

The interpretation of the polymorphic λ-calculus has been one of the most serious challenges in the semantics of programming languages. Of course,

[1] We could also define the polymorphic λ-calculus more generally relative to a theory T, but the discussion here is based on using the empty theory.

it is possible to construct a term model for the calculus as we did earlier for the simply-typed λ-calculus. But finding a model analogous to the full type frame is much harder. To appreciate the primary reason for this difficulty, let us attempt to provide such a model by partial analogy to the one we used for ML_0. We will need the notion of a *type-value environment* ι that maps type variables to semantic interpretations as sets. The interpretation $[\![t]\!]$ of a type t is a function that takes type assignments indicating the meanings of free variables of t into sets. As with the full type frame, we define $[\![s \to t]\!]\iota = [\![s]\!]\iota \to [\![t]\!]\iota$ where the arrow on the right is the full function space operator. The central question is, how do we interpret $\Pi a.\ t$? Let us naively take this to be a product of sets indexed over sets; an element of $[\![\Pi a.\ t]\!]\iota$ is a function that associates with each set X an element of the set $[\![t]\!]\iota[a \mapsto X]$. Such a function is called a *section* of the *indexed family* $X \mapsto [\![t]\!]\iota[a \mapsto X]$. It can improve the readability of expressions involving sections to write the application of a section to a set with the argument as a subscript. So, for example, if π is a section of $X \mapsto [\![t]\!]\iota[a \mapsto X]$, then $\pi_X \in [\![t]\!]\iota[a \mapsto X]$. To provide the semantic interpretation for terms, we will also want to know that a form of substitution lemma holds for types: $[\![[s/a]t]\!]\iota = [\![t]\!]\iota[a \mapsto [\![s]\!]\iota]$.

Given a type-value environment ι and a type assignment H, let us say that ρ is an ι, H*-environment* if $\rho(x) \in [\![H(x)]\!]\iota$ for each $x \in H$. If $H \vdash M : t$, then the interpretation $[\![H \rhd M : t]\!]$ is a function that takes a type-value environment ι and an ι, H-environment as an argument and returns a value in $[\![t]\!]\iota$. It sometimes helps to drop the type information in the interpreted expression and write $[\![M]\!]\iota\rho$ to save clutter when the types are clear enough from context.

The interpretation of terms is now defined by induction on their structure. The interpretation of an abstraction $[\![H \rhd \lambda x : s.\ M : s \to t]\!]\iota\rho$ over term variables is the function from $[\![s]\!]\iota$ to $[\![t]\!]\iota$ defined by

$$d \mapsto [\![H, x : s \rhd M : t]\!]\iota(\rho[x \mapsto d]).$$

On the other hand, the interpretation of the application of a term to a term is given by the usual application of a function to its argument: $[\![M(N)]\!]\iota\rho = ([\![M]\!]\iota\rho)([\![N]\!]\iota\rho)$. In considering the interpretation of the application of a term $M : \Pi a.\ s$ to a type t, recall that $[\![H \rhd M : \Pi a.\ s]\!]\iota\rho$ is a section of the indexed family $X \mapsto [\![s]\!]\iota[a \mapsto X]$. We take $[\![H \rhd M\{t\} : [t/a]s]\!]\iota\rho = ([\![H \rhd M : \Pi a.\ s]\!]\iota\rho)_X$ where $X = [\![t]\!]\iota$. This squares with the claim that $[\![[t/a]s]\!]\iota = [\![s]\!]\iota[a \mapsto [\![t]\!]\iota]$. Now, finally, the meaning of a type abstraction $[\![H \rhd \Lambda a.\ M : \Pi a.\ t]\!]\iota\rho$ is a section of the indexed family $X \mapsto [\![t]\!]\iota[a \mapsto X]$ given by $X \mapsto [\![H \rhd M : t]\!](\iota[a \mapsto X])\rho$. One must show that ρ is an $\iota[a \mapsto X], H$-value environment, but this follows from the restriction in [Π-Intro] that says the type variable a does not appear in H.

The semantics just sketched is sufficiently simple and convincing that something like it was actually used in early discussions of the semantics of the calculus [Reynolds, 1983]. As it stands, however, there is a problem with the interpretation of types. A type is presumably a function from type-value environments to sets. But consider a type like $\Pi a.\ a$, which we have naively interpreted as the family of sections of the indexed family $X \mapsto X$. In other words, $[\![\Pi a.\ a]\!]\iota$ is the 'product' of *all* sets. We must assume that this product is itself a set, because this is needed to make our interpretation work. Consider, for instance, the term $M = (\Lambda a.\ \lambda x : a.\ x)(\Pi a.\ a)$. The term $\Lambda a.\ \lambda x : a.\ x$ is interpreted as a section over all sets and M is interpreted as the application of this section to the meaning of $\Pi a.\ a$.

This leads us to a foundational question concerning what collections are considered to be sets. One of the crucial discoveries of logicians in the late nineteenth and early twentieth centuries was the fact that care must be taken in how collections of entities are formed if troubling paradoxes are to be avoided. Such paradoxes caused intricate and carefully constructed theories of the foundations of mathematics to collapse into nonsense. Perhaps the best-known and most important of these paradoxes is *Russell's paradox,* which is named after the philosopher and logician Bertrand Russell. It can be described quite simply as follows. Let us assume that any property at all can be used to define a collection of entities. Define $\mathcal{X}$ to be the collection of all collections X having the property that X is not an element of X. This seems clear enough since we (think we) know what it means for an entity to be part of a collection. Let us therefore ask whether $\mathcal{X}$ is in $\mathcal{X}$ or not. Well, if it is, then it has the property common to all elements of $\mathcal{X}$, that of not being a member of itself. This is a contradiction, since we had postulated that $\mathcal{X}$ was a member of itself. Suppose, on the other hand, that $\mathcal{X}$ is *not* a member of itself. Then this is a contradiction as well, since we defined $\mathcal{X}$ to be those collections having exactly this property.

Returning now to the polymorphic λ-calculus and our problem with its interpretation, we must avoid a transgression into Russell's paradox. Technically the problem is that the restrictions placed on the formation of sets makes it impossible to view the product of all sets as itself a set. However, the underlying phenomenon here was recognized by Russell and by the mathematician and philosopher of science Henri Poincaré in a property he called 'impredicativity'. If a set $\mathcal{X}$ and an entity X are defined so that X is a member of $\mathcal{X}$ but is defined only by reference to $\mathcal{X}$, then the definition of $\mathcal{X}$ or X is said to be *impredicative.* Clearly this is the case for Russell's paradox, but it also applies to the class $\mathcal{X}$ of semantic domains that we are attempting to use for modeling the types of the polymorphic λ-calculus and the domain X that is to serve as the interpretation for $\Pi a.\ a$. That this problem has no resolution when one is dealing with *sets* was shown by Reynolds [Reynolds, 1984] (a more refined treatment appears in [Reynolds and Plotkin, 1990]; thus we must look for another class of semantics domains with which to interpret our types.

7.5 Simple types as PER's.

Recall the problem cited earlier with the use of inclusive predicates to model types: namely that the ξ-rule is not satisfied. An approach to solving this problem is to use an equivalence relation on subsets of the universal domain to induce the needed equalities. In particular:

Definition 7.5.1. Let A be a set. A *Partial Equivalence Relation (PER)* on A is a relation $R \subseteq A \times A$ that is transitive and symmetric. The *domain* of a partial equivalence relation R is the set $\mathrm{dom}(R) = \{a \in A \mid a \; R \; a\}$.

We write $a \; R \; b$ to mean that $(a, b) \in R$. Note that if R is a PER on a set A and $a \; R \; b$ for any $a, b \in A$, then a and b are in the domain of A by the axioms.

Now, suppose we are given a model of the untyped λ-calculus. Say $\Phi : U \rightarrow [U \rightarrow U]$ and $\Psi : [U \rightarrow U] \rightarrow U$ satisfy $\Phi \circ \Psi = \mathbf{id}_{[U \,\rightarrow\, U]}$. Let X be any PER on U. Take $\mathcal{P}[\![\mathbf{o}]\!] = X$ and define PER interpretations for types by structural induction as follows. Suppose $\mathcal{P}[\![s]\!]$ and $\mathcal{P}[\![t]\!]$ are PER's, then

$$\begin{array}{c} f \; (\mathcal{P}[\![s \rightarrow t]\!]) \; g \\ \text{iff} \\ \text{for each } d \text{ and } e,\ d \; (\mathcal{P}[\![s]\!]) \; e \text{ implies } \Phi(f)(d) \; (\mathcal{P}[\![t]\!]) \; \Phi(g)(e). \end{array} \tag{7.1}$$

It is easy to check that each of these relations is a partial equivalence. To see how they interpret terms, we use a semantic function that gives untyped meaning to typed terms. Given a term M of the simply-typed λ-calculus, let $\mathcal{U}[\![M]\!]$ be the meaning of untyped λ-term $\mathrm{erase}(M)$ in U based on the pair Φ, Ψ. Given a PER R on U and an element $d \in \mathrm{dom}(R)$, let $[d]_R$ be the equivalence class of d relative to R, that is, $[d]_R = \{e \mid d \; R \; e\}$.

Let M be a term of the typed λ-calculus such that $H \vdash M : t$. The meaning of M is given relative to a function ρ from variables $x \in H$ into U such that $\rho(x)$ is in the domain of the relation $\mathcal{P}[\![H(x)]\!]$ for each $x \in H$. Such a function is called an H-environment for the PER interpretation. Now, the interpretation for the term M is quite simple:

$$\mathcal{P}[\![H \rhd M : t]\!]\rho = [\mathcal{U}[\![M]\!]\rho]_{\mathcal{P}[\![t]\!]}$$

There are two basic facts to be proved about this interpretation. First, if M has type t then the interpretation of M is in the domain of the relation $\mathcal{P}[\![t]\!]$. Second, all of the equational rules of the typed λ-calculus are satisfied. Establishing these properties involves proving slightly more general facts. Given H-environments ρ and θ, define $\rho \sim_H \theta$ if, for each $x \in H$, $\rho(x) \; \mathcal{P}[\![H(x)]\!] \; \theta(x)$.

Lemma 7.5.2. *Suppose $H \vdash M : t$. If ρ and θ are H-environments and $\rho \sim_H \theta$, then $(\mathcal{U}[\![M]\!]\rho)\, \mathcal{P}[\![t]\!]\, (\mathcal{U}[\![M]\!]\theta)$.*

Proof. The proof is by induction on the structure of M. If $M \equiv x$, then $x \in H$, $t \equiv H(x)$, and $\rho \sim \theta$ implies the desired conclusion.

Case $M \equiv \lambda x : u.\ M'$ where $t \equiv u \to v$. Suppose $d\, \mathcal{P}[\![u]\!]\, e$. Then

$$\Phi(\mathcal{U}[\![\lambda x : u.\ M']\!]\rho)(d) = \Phi(\Psi(d \mapsto \mathcal{U}[\![M']\!]\rho[x \mapsto d])(d) = \mathcal{U}[\![M']\!]\rho[x \mapsto d],$$

and, similarly, $\Phi(\mathcal{U}[\![\lambda x : u.\ M']\!]\theta)(e) = \mathcal{U}[\![M']\!]\theta[x \mapsto e]$. Now,

$$(\mathcal{U}[\![M']\!]\rho[x \mapsto d])\, \mathcal{P}[\![v]\!]\, (\mathcal{U}[\![\lambda x : u.\ M']\!]\theta[x \mapsto e])$$

by the inductive hypothesis. The desired conclusion therefore follows from the definition of $\mathcal{P}[\![t]\!]$.

Case $M \equiv L(N)$ where $H \vdash L : s \to t$ and $H \vdash N : s$. By the inductive hypothesis for N, $(\mathcal{U}[\![N]\!]\rho)\, \mathcal{P}[\![s]\!]\, (\mathcal{U}[\![N]\!]\theta)$. So, by the inductive hypothesis for L, $d\ \mathcal{P}[\![t]\!]\ e$ where $d = \Phi(\mathcal{U}[\![L]\!]\rho)(\mathcal{U}[\![N]\!]\rho) = \mathcal{U}[\![M]\!]\rho$ and $e = \Phi(\mathcal{U}[\![L]\!]\theta)(\mathcal{U}[\![N]\!]\theta) = \mathcal{U}[\![M]\!]\theta$. ∎

Corollary 7.5.3. *If $H \vdash M : t$ and ρ is an H-environment, then $\mathcal{P}[\![H \rhd M : t]\!]\rho$ is in the domain of the relation $\mathcal{P}[\![t]\!]$.*

Lemma 7.5.4. *If $\vdash (H \rhd M' = N' : t')$ and $\rho \sim_H \theta$ are H-environments, then*

$$\mathcal{P}[\![H \rhd M' : t']\!]\rho = \mathcal{P}[\![H \rhd N' : t']\!]\theta.$$

Proof. The proof is by induction on the height of the derivation of the judgement $\vdash (H \rhd M' = N' : t')$. By way of illustration, let us consider the case in which the last step of the proof is an instance of the ξ-rule. Suppose the last step of the derivation is an instance of

$$\{\xi\} \qquad \frac{T \vdash (H,\ x : s \rhd M = N : t)}{T \vdash (H \rhd \lambda x : s.\ M = \lambda x : s.\ N : s \to t)}$$

where $M' \equiv \lambda x : s.\ M$ and $N' \equiv \lambda x : s.\ N$ and $t' \equiv s \to t$. By the inductive hypothesis, $(\mathcal{U}[\![M]\!]\rho[x \mapsto d])\ \mathcal{P}[\![t]\!]\ (\mathcal{U}[\![N]\!]\theta[x \mapsto e])$ whenever $e\ \mathcal{P}[\![s]\!]\ d$. Hence $(\mathcal{U}[\![\lambda x.\ M]\!]\rho)\ \mathcal{P}[\![s \to t]\!]\ (\mathcal{U}[\![\lambda x.\ N]\!]\theta)$ by the definition of the PER $\mathcal{P}[\![s \to t]\!]$. Thus $\mathcal{P}[\![H \rhd \lambda x : s.\ M : s \to t]\!]\rho = \mathcal{P}[\![H \rhd \lambda x : s.\ N : s \to t]\!]\theta$. ∎

Corollary 7.5.5. *If $\vdash (H \rhd M = N : t)$, then $\mathcal{P}[\![H \rhd M : t]\!] = \mathcal{P}[\![H \rhd N : t]\!]$.*

7.6 PER's as a model of polymorphic types.

We have seen that if we are given a model (U, Φ, Ψ) of the untyped λ-calculus, then PER's over U can be used to interpret the types of the simply-typed λ-calculus; now we consider how PER's can be used to interpret types of the polymorphic λ-calculus. To make this extension, we define the meaning of a type relative to a type-value environment that maps type variables to PER's. To interpret $\Pi a.\, t$ as an indexed family over PER's, let ι be a type-value environment that has all of the free type variables of $\Pi a.\, t$ in its domain. We want to define its meaning to be the 'product' of the relations $\mathcal{P}[\![t]\!]\iota[a \mapsto R]$ as R ranges over the PER's over U. Given $x, y \in U$, this says that x and y are related modulo $\mathcal{P}[\![t]\!]\iota[a \mapsto R]$ for each R. That is,

$$\mathcal{P}[\![\Pi a.\, t]\!]\iota = \bigcap_{R \in \mathrm{PER}} \mathcal{P}[\![t]\!]\iota[a \mapsto R]. \tag{7.2}$$

This defines a partial equivalence relation because the intersection of PER's is a PER. Notice the role of impredicativity in Equation 7.2 where the intersection ranges over the class of all PER's. Of course, the PER that interprets $\Pi a.\, t$ itself is in this collection, but there is no problem with existence in this case, because the intersection of a set of sets is again a set. The interpretation of function spaces is given same as before in 7.1.

The *erasure* of a term of the polymorphic λ-calculus is defined by induction as follows:

- $\mathrm{erase}(x) \equiv x$
- $\mathrm{erase}(\lambda x : t.\, M) \equiv \lambda x.\, \mathrm{erase}(M)$
- $\mathrm{erase}(M(N)) \equiv (\mathrm{erase}(M))(\mathrm{erase}(N))$
- $\mathrm{erase}(\Lambda a.\, M) \equiv \mathrm{erase}(M)$
- $\mathrm{erase}(M\{t\}) \equiv \mathrm{erase}(M)$

The meaning of a term is defined using the meaning, as an untyped term, of its erasure. We assume that a retraction from U onto $[U \to U]$ is given and define $\mathcal{U}[\![M]\!]$ to be the meaning of untyped λ-term $\mathrm{erase}(M)$ in U. As before, given a PER R on U and an element $d \in \mathrm{dom}(R)$, let $[d]_R$ be the equivalence class of d relative to R, that is, $[d]_R = \{e \mid d\ R\ e\}$. Let M be a term of the polymorphic λ-calculus such that $H \vdash M : t$. The meaning of M is given relative to a type-value environment ι and a function ρ from variables $x \in H$ into U such that $\rho(x)$ is in the domain of the relation $\mathcal{P}[\![H(x)]\!]\iota$. The interpretation of the term M is given by

$$\mathcal{P}[\![M]\!]\iota\rho = [\,\mathcal{U}[\![M]\!]\rho\,]_{\mathcal{P}[\![t]\!]\iota}.$$

To complete the demonstration that this defines a model, it must be shown that if a term M has type t, then the interpretation of M relative to an H, ι-environment is in the domain of the relation $\mathcal{P}[\![t]\!]\iota$, and that the equational rules of the polymorphic λ-calculus are satisfied. The treatment follows the general pattern of the argument for interpretation of simple types. We start with the following basic:

Lemma 7.6.1. $\mathcal{P}[\![[s/a]t]\!]\iota = \mathcal{P}[\![t]\!]\iota[a \mapsto \mathcal{P}[\![s]\!]\iota]$.

Lemma 7.6.2. *Suppose* $H \vdash M : t$ *and* ρ, θ *are* H, ι*-environments such that*

$$\rho(x)\ (\mathcal{P}[\![H(x)]\!]\iota)\ \theta(x)$$

for each $x \in H$. *Then*

$$(\mathcal{U}[\![M]\!]\rho)\ (\mathcal{P}[\![t]\!]\iota)\ (\mathcal{U}[\![M]\!]\theta).$$

Proof. The proof is by induction on the structure of M. Let us look at the cases for type abstraction and application. If $M \equiv \Lambda a.\ M' : \Pi a.\ t'$, then

$$(\mathcal{U}[\![M']\!]\rho)\ (\mathcal{P}[\![t']\!]\iota[a \mapsto R])\ (\mathcal{U}[\![M']\!]\theta)$$

for any PER R by the inductive hypothesis. Since $\mathrm{erase}(M') \equiv \mathrm{erase}(M)$ and the interpretation of $\Pi a.\ t'$ is the intersection of PER's of the form $\mathcal{P}[\![t']\!]\iota[a \mapsto R]$, we must have

$$(\mathcal{U}[\![M]\!]\rho)\ (\mathcal{P}[\![\Pi a.\ t']\!]\iota)\ (\mathcal{U}[\![M]\!]\theta).$$

Suppose now that $M \equiv M'\{s\}$. Then $H \vdash M' : \Pi a.\ t'$ and $t \equiv [s/a]t'$. Applying the inductive hypothesis to M', we have

$$(\mathcal{U}[\![M']\!]\rho)\ (\bigcap_{R \in PER} \mathcal{P}[\![t']\!]\iota[a \mapsto R])\ (\mathcal{U}[\![M']\!]\theta)$$

and therefore, in particular,

$$(\mathcal{U}[\![M']\!]\rho)\ (\mathcal{P}[\![t']\!]\iota[a \mapsto \mathcal{P}[\![s]\!]\iota])\ (\mathcal{U}[\![M']\!]\theta)$$

Now $\mathrm{erase}(M') \equiv \mathrm{erase}(M)$ so, by Lemma 7.6.1,

$$(\mathcal{U}[\![M]\!]\rho)\ (\mathcal{P}[\![[s/a]t']\!]\iota)\ (\mathcal{U}[\![M]\!]\theta).$$

∎

Corollary 7.6.3. *If* $H \vdash M : t$ *and* ρ *is an* H, ι*-environment, then* $\mathcal{U}[\![H \rhd M : t]\!]\iota\rho$ *is in the domain of the relation* $\mathcal{P}[\![t]\!]\iota$.

Lemma 7.6.4. *If* $\vdash (H \rhd M' = N' : t')$ *and* ρ, θ *are* H, ι*-environments such that* $\rho(x)\ (\mathcal{P}[\![H(x)]\!]\iota)\ \theta(x)$ *for each* $x \in H$*, then*

$$\mathcal{P}[\![H \rhd M' : t']\!]\iota\rho = \mathcal{P}[\![H \rhd N' : t']\!]\iota\theta.$$

Corollary 7.6.5. *If* $\vdash (H \rhd M = N : t)$*, then* $\mathcal{P}[\![H \rhd M : t]\!] = \mathcal{P}[\![H \rhd N : t]\!]$*.*

8 Conclusion

A more detailed treatment of the topics in this chapter, including complete proofs of most of the theorems, can be found in [Gunter, 1992]; a great deal of further material on the semantics of types can be found in other chapters of this handbook. *There is much more that can be said about each of the models* described in the sections here: for example, since seminal work of Bruce and Longo [Bruce and Longo, 1990], PER's have been quite successful as a model of *subtypes* as well as parametric polymorphism and, as one can glean from the references in [Scedrov, 1990], we have only just scratched the surface of what can be said about PER's as a model of parametric polymorphism. *There are other models* besides the ones that have been covered here: a whole subject of the semantics of types as *categorical objects* has been omitted (see [Poigné, 1992]) and there is a rapidly evolving theory of types as *games* [Abramsky *et al.*, 1993]. And, particularly, *there are hosts of type systems* that have not been discussed in this chapter. One large category of these only touched upon here is that of types for *object-oriented* programming languages. The reader can pursue this topic further through [Gunter and Mitchell, 1994; Palsberg and Schwartzback, 1993] and the references there. A second is the semantics of recursive types, whose importance was hinted at in Section 2 but not discussed in any detail in the remainder of the chapter; the reader can pursue the topic further by consulting [Gunter, 1992]. A third important class of type systems is those that are used for *modules*; two references that can be used as a starting point are [Harper and Mitchell, 1993] and [Moggi, 1991].

I would like to acknowledge Samson Abramsky for encouraging me to undertake this discussion of the semantics of types. I also thank Sandip Biswas, Roy Crole, and Narciso Martí-Oliet for their help in proof-reading drafts of the work.

References

[Abelson and Sussman, 1985] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, 1985.

[Abramsky and Jung, 1994] S. Abramsky and A. Jung. Domains. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Theoretical Computer Science*, pages 1–190. Oxford University Press, 1994.

[Abramsky *et al.*, 1993] S. Abramsky, R. Jagadeesan, and P. Malacaria. Games and full abstraction for PCF II: Second preliminary announcement. Manuscript, 1993.

[Barendregt, 1992] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Theoretical Computer Science, Volume 2 Background: Computational Structures*, pages 117–310. Oxford University Press, 1992.

[Berry *et al.*, 1985] G. Berry, P.-L. Curien, and J.-J. Levy. Full abstraction for sequential languages: the state of the art. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*, pages 89–132. Cambridge University Press, 1985.

[Berry, 1978] G. Berry. Stable models of typed λ-calculus. In *International Colloquium on Automata, Languages and Programs*, pages 72–89. *Lecture Notes in Computer Science vol. 62,* Springer, 1978.

[Berry, 1979] G. Berry. *Modèles Complètement Adéquats et Stables des Lambda-calculs Typés.* Thèse d'État, University of Paris VII, 1979.

[Breazu-Tannen *et al.*, 1990] V. Breazu-Tannen, C. Gunter, and A. Scedrov. Computing with coercions. In M. Wand, editor, *Lisp and Functional Programming,* pages 44–60. ACM, 1990.

[Bruce and Longo, 1990] K. B. Bruce and G. Longo. A modest model of records, inheritance, and bounded quantification. *Information and Computation*, 87:196–240, 1990.

[Burn *et al.*, 1986] G. L. Burn, C. Hankin, and S. Abramsky. Strictness analysis for higher-order functions. *Science of Computer Programming*, 7:249–278, 1986.

[Cardelli, 1987] L. Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8:147–172, 1987.

[Cardelli, 1988] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[Clément *et al.*, 1986] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: mini-ML. In *Symposium on LISP and Functional Programming,* pages 13–27. ACM, 1986.

[Damas and Milner, 1982] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Principles of Programming Languages*, pages 207–212. ACM, 1982.

[Damas, 1985] L. Damas. *Type Assignment in Programming Languages.* PhD thesis, Edinburgh University, 1985.

[Duba *et al.*, 1991] B. Duba, R. Harper, and D. B. MacQueen. Typing first-class contiuations in ML. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 245–254. ACM, 1991.

[Felleisen and Friedman, 1986] M. Felleisen and D. P. Friedman. Control operators, the secd-machine, and the λ-calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. North Holland, 1986.

[Friedman, 1975] H. Friedman. Equality between functionals. In R. Parikh, editor, *Proceedings of the Logic Colloqium '73*, pages 22–37. *Lecture Notes in Mathematics vol. 453,* Springer, 1975.

[Girard, 1972] J. Y. Girard. *Interprétation Fonctionelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur.* Thèse d'État, University of Paris VII, 1972.

[Gunter and Mitchell, 1994] C. A. Gunter and J. C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design.* The MIT Press, 1994.

[Gunter and Scott, 1990] C. A. Gunter and D. S. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 633–674. North-Holland, 1990.

[Gunter, 1992] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing. The MIT Press, 1992.

[Gunter, 1993] C. A. Gunter. Forms of semantic specification. In B. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science: Essays and Tutorials*, pages 332–353. World Scientific Publishers, 1993.

[Harper and Mitchell, 1993] R. Harper and J. C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 1993. To appear.

[Hindley and Seldin, 1986] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-calculus.* Cambridge University Press, 1986.

[Howard, 1973] W. Howard. Hereditarily majorizable functionals of finite type. In A. S. Troelstra, editor, *Metamathematical Investigation of Intuitionistic*

Arithmetic and Analysis, pages 454–461. *Lecture Notes in Mathematics vol 344,* Springer, 1973.

[IEE, 1991] IEEE Computer Society, New York. *IEEE Standard for the Scheme Programming Language*, IEEE standard 1178-1990 edition, 1991.

[Jim and Meyer, 1991] T. Jim and A. R. Meyer. Full abstraction and the context lemma. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 131–151. Springer-Verlag, September 1991.

[Kahn, 1987] Gilles Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer-Verlag, 1987.

[Leroy, 1993] X. Leroy. Polymorphism by name for references and continuations. In S. L. Graham, editor, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 220–231. ACM, 1993.

[MacQueen *et al.*, 1986] D. B. MacQueen, G. D. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.

[Martin-Löf, 1971] Per Martin-Löf. An intuitionistic theory of types. unpublished, 1971.

[Meyer, 1982] A. R. Meyer. What is a model of the lambda calculus? *Information and Control*, 52:87–122, 1982.

[Milner and Tofte, 1991] R. Milner and M. Tofte. *Commentary on Standard ML.* The MIT Press, 1991.

[Milner *et al.*, 1990] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[Milner, 1978] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mitchell, 1990] J. C. Mitchell. Types systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 365–458. North-Holland, 1990.

[Moggi, 1991] E. Moggi. A category-theoretic account of program modules. *Mathematical Structures in Computer Science*, 1:103–139, 1991.

[Palsberg and Schwartzback, 1993] J. Palsberg and M. Schwartzback. *Object-Oriented Type Systems.* Wiley, 1993.

[Plotkin, 1976] G. D. Plotkin. A powerdomain construction. *SIAM Journal of Computing*, 5:452–487, 1976.

[Plotkin, 1981] G. D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, Computer Science Department, Aarhus University, Ny Munkegade—DK 8000 Aarhus C—Denmark, 1981.

[Poigné, 1992] A. Poigné. Basic category theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Theoretical Computer Science, Volume 1 Background: Mathematical Structures*, pages 413–640. Oxford University Press, 1992.

[Reynolds and Plotkin, 1990] J. C. Reynolds and G. D. Plotkin. On functors expressible in the polymorphic typed lambda calculus. In Gérard Huet, editor, *Logical Foundations of Functional Programming*, University of Texas at Austin Year of Programming, pages 127–152. Addison-Wesley, Reading, Massachusetts, 1990.

[Reynolds, 1974] J. C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, pages 408–425. *Lecture Notes in Computer Science vol. 19,* Springer, 1974.

[Reynolds, 1980] J. C. Reynolds. Using category theory to design implicit conversions and generic operators. In N. D. Jones, editor, *Semantics-Directed Compiler Generation*, pages 211–258. *Lecture Notes in Computer Science vol. 94,* Springer, 1980.

[Reynolds, 1983] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers B. V. (North-Holland).

[Reynolds, 1984] J. C. Reynolds. Polymorphism is not set-theoretic. In G. Kahn, D. B. MacQueen, and G. D. Plotkin, editors, *Semantics of Data Types*, pages 145–156. *Lecture Notes in Computer Science vol. 173,* Springer, 1984.

[Robinson, 1965] J. A. Robinson. A machine oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[Scedrov, 1990] A. Scedrov. A guide to polymorphic types. In P. Odifreddi, editor, *Logic and Computer Science*, pages 387–420. Academic Press, 1990.

[Scott, 1969] D. S. Scott. A type theoretical alternative to CUCH, ISWIM, OWHY. Unpublished manuscript, 1969, 1969.

[Scott, 1976] D. S. Scott. Data types as lattices. *SIAM Journal of Computing*, 5:522–587, 1976.

[Scott, 1981] D. S. Scott. Some ordered sets in computer science. In I. Rival, editor, *Ordered Sets*, pages 677–718. D. Reidel, 1981.

[Scott, 1982a] D. S. Scott. Domains for denotational semantics. In M. Nielsen and E. M. Schmidt, editors, *International Colloquium on Automata, Languages* *and Programs*, pages 577–613. *Lecture Notes in Computer Science vol. 140,* Springer, 1982.

[Scott, 1982b] D. S. Scott. Lectures on a mathematical theory of computation. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology,* pages 145–292. *NATO Advanced Study Institutes Series,* D. Reidel, 1982.

[Sokolowksi, 1991] S. Sokolowksi. *Applicative High Order Programming: The standard ML perspective.* Chapman and Hall, 1991.

[Statman, 1982] R. Statman. Completeness, invariance and λ-definability. *Journal of Symbolic Logic*, 47:17–26, 1982.

[Statman, 1985a] R. Statman. Equality between functionals. In *Harvey Friedman's Research on the Foundations of Mathematics*, pages 331–338. North-Holland, 1985.

[Statman, 1985b] R. Statman. Logical relations and the typed λ-calculus. *Information and Control*, 65:85–97, 1985.

[Statman, 1986] R. Statman. On translating lambda terms into combinators: the basis problem. In A. Meyer, editor, *Symposium on Logic in Computer Science*, pages 378–382. ACM, 1986.

[Stoughton, 1991] A. Stoughton. Interdefinability of parallel operations in PCF. *Theoretical Computer Science*, 79:357–358, 1991.

[Tait, 1967] W. W. Tait. Intensional interpretation of functionals of finite type. *Journal of Symbolic Logic*, 32:198–212, 1967.

[Tennent, 1992] B. Tennent. Denotational semantics. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Theoretical Computer Science.* Oxford University Press, 1992.

[Tofte, 1990] M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, 1990.

[Wright, 1993] A. K. Wright. Polymorphism for imperative languages without imperative types. Technical Report COMP TR93-200, Department of Computer, Rice University, 1993.