INFORMATION AND COMPUTATION 76, 278-346 (1988)

Pebble, a Kernel Language for Modules and Abstract Data Types*

B. LAMPSON† AND R. BURSTALL‡

†*Systems Research Center, Digital Equipment Corporation, 130 Lytton Ave., Palo Alto, California 94301; and*
‡*Department of Computer Science, University of Edinburgh, Kings Buildings, Mayfield Road, Edinburgh EH9 3JZ, Scotland*

A small set of constructs can simulate a wide variety of apparently distinct features in modern programming languages. Using a kernel language called Pebble based on the typed lambda calculus with bindings, declarations, dependent types, and types as compile time values, we show how to build modules, interfaces and implementations, abstract data types, generic types, recursive types, and unions. Pebble has a concise operational semantics given by inference rules.

1. INTRODUCTION

Programming language designers have invented a number of features to support the writing of large programs in a modular way which takes advantage of type-checking. As languages have grown in size these features have been added to the basic structure of expressions, statements, and procedures in various ad hoc fashions, increasing the syntactic and semantic complexity of the language. It is not very clear what the underlying concepts or the language design options are. In particular cases various kinds of parameterised types or modules are offered, and it is unclear how these are related to the ideas of function definition and application, which can be formalised very simply in the lambda calculus.

This paper describes a relatively small programming language called Pebble, which provides a precise model for these features. It is a functional language, based upon the lambda calculus with types. It is addressed to the problems of data types, abstract data types, and modules. It also deals with

* This work was supported in part by the Xerox Palo Alto Research Center. An earlier version was presented at the International Symposium on Semantics of Data Types in Sophia-Antipolis, France, in June of 1984, and appears in the proceedings of that symposium, "Lecture Notes in Computer Science, Vol 173" (G. Kahn, D. B. MacQueen, and G. Plotkin, Eds.), pp. 1-50, Springer-Verlag, Berlin/New York.

0890-5401/88 $3.00

the idea of generic values. It does not reflect all aspects of programming languages, since we have not dealt with assignment, exceptions, or concurrency, although we believe that these could be added to our framework. Our intention is that it should be possible to express the semantics of a sizeable part of a real programming language by giving rules which rewrite it into Pebble. This follows the method used by Bauer and his colleagues (Bauer *et al.*, 1978) to express the semantics of their wide spectrum language. We were particularly concerned with the Cedar language (an extension of Mesa (Mitchell *et al.*, 1979)) which is in use at Xerox PARC. One of us has defined the quite complex part of this language which is concerned with data types and modules in terms of rewrite rules which convert Cedar to an earlier version of Pebble (Lampson, 1983).

An earlier version of this paper appeared as (Burstall and Lampson, 1984). In revising it we have

- provided a better treatment of union types;
- introduced the notion of "extended type" which enables us to carry around the operations appropriate to a value as part of its type;
- introduced a type constructor "⊗" which enables one to apply a polymorphic function without giving an explicit type argument;
- introduced a notion of inclusion between bindings so that a module can accept a bigger binding than the one it needs;
- introduced a coercion mechanism to implement these last two features;
- corrected a mistake in the semantics of recursion.

We have also removed a number of minor errors and infelicities. These changes in the direction of practicality have enlarged somewhat the original small language which was intended primarily to explicate the concepts of modules and system modelling. The original simplicity may still be discerned with the eye of faith.

This paper was mostly written in 1983. (Refereeing and revising the Journal version took some time.) Since then the world has moved along and people have become much more familiar with existential and universal dependent types due to the growing appreciation of Martin-Lof's work on type theory. If we were rewriting it today we might be more concise and less pedagogical in the first part of the paper. None the less our aims have been somewhat different from those of most people concerned with type theory. Starting from our original concern with the module structure of Cedar, we are now trying to design a language which could be a firm basis for a practical system programming language. We have kept our formal semantics fairly close to a possible practical implementation.

Recently we have made some simplifications in Pebble, profiting from work by Luca Cardelli (1986) on a Pebble-like language with a denotational semantics. We have a version of the language with simpler operational semantics and we have added exceptions and assignment. However, it is a sizeable task to write a new paper on the basis of our new formal semantics, so we have decided to publish the present version in the meantime.

The first part of this paper is informal with examples, addressed to the language designer or user. The reader may or may not wish to dig into the precise semantic definition in Sections 4 and 5. For a less detailed exposition of Pebble stressing motivation see Burstall (1984).

Practical Motivation

A principal idea which we wish to express in our formalism is the linking together of a number of modules into a large program. This may be summarized as follows: Each program module produces an *implementation* of some collection of data types and procedures. In order to do so it may require the implementations supplied to it by some other modules. This traffic in implementations is controlled by *interfaces* which say what kind of implementation is required or produced by a module. These interfaces name the data types and specify the argument and result types of the procedures. Given a large collection of modules, perhaps the work of many people at different times, it is essential to be able to express easily different ways of connecting them together, that is, ways of providing the implementations needed by each module. An input interface of a module may be satisfied by the implementations produced by several different modules or different "versions" of the same module.

We believe that linking should not be described in a primitive and ad hoc special purpose language; it deserves more systematic treatment. In our view the linking should be expressed in a functional applicative language, in which modules are regarded as *functions* from implementations to implementations. Furthermore this language should be typed, and the interfaces should play the role of types for the implementations. Thus we have the correspondence

$$\text{implementation} \leftrightarrow \text{value}$$

$$\text{interface} \leftrightarrow \text{type}$$

$$\text{module} \leftrightarrow \text{function}.$$

Function application is more appropriate for linking than schemes based on the names of the modules and the sequence in which they are presented. By choosing suitable structured types in a functional language we can get a

simple notation for dealing with "big" objects (pieces of a program) as if they were "small" ones (numbers); this is the basic good trick in matrix algebra. Thus we hope to make "Programming in the Large" look very much like "Programming in the Small."

Another advantage of this approach to linking is that the linking language can be incorporated in the programming language. We hope in this way to achieve both conceptual economy and added flexibility in expressing linking. By contrast, the usual approach to the linking problem, exemplified by Mesa and C-Mesa (Mitchell *et al.* 1979), has a programming language (Mesa) with a separate and different linking language (C-Mesa) which sits on top of it so to speak. The main advantage of this approach is that a separate linking language can be used for linking modules of more than one programming language, although in the past this advantage has been gained only at the price of using an extremely primitive linking language.

A linking system called the System Modeller was built by Eric Schmidt for his Ph. D. thesis work, supervised by one of us (B.L.). He used an earlier version of Pebble with some modifications, notably to provide default values for arguments since these are often obvious from the context (Schmidt, 1982; Lampson and Schmidt, 1983). The System Modeller was used several people to build large systems, but the implementation has not been polished sufficiently for widespread use.

Our other practical motivation was to investigate how to provide *polymorphic* functions in Cedar, that is ones which will work uniformly for argument values of different types; for example, a matrix transpose procedure should work for integer matrices as well as for real matrices.

There have been two experimental implementations of Pebble, one by Glenn Stone at Manchester University in Prolog, and one by Hugh Stabler at Edinburgh University in ML. These were both student projects. There have been several other partial implementations.

Outline of the Paper

We start from Landin's view of programming languages as lambda calculus sweetened with syntactic sugar (Landin, 1964). Since we are dealing with typed languages, we must use typed lambda calculus, but it turns out that we need to go further and extend the type system with dependent types. We take types as values, although they need to be handled only during type-checking (which may involve some evaluation) and not at execution time. We thus handle all variable binding with just one kind of lambda expression, as opposed to Reynolds (1974). Another extension is needed because, while procedures accept n-tuples of values, for example, (1, 5, 3), at the module level it is burdensome to rely on position in a sequence to identify parameters and it is usual to associate them with

names, for example ($x \sim 1$, $y \sim 5$, $z \sim 3$). This leads to the notion of a *binding*. To elucidate the notion of a parameterised module we include such bindings as values in Pebble. It turns out that the scoping of the names which they contain does not create problems.

To define a precise meaning for Pebble programs we give an operational semantics in the form of inference rules, using a formalism due to Plotkin (1981), with some variations. We could have attempted a denotational semantics, but this would have raised theoretical questions rather different from our concerns about language design. As far as we know it would be quite possible to give a satisfactory denotational semantics for Pebble. Cardelli (1986) gives a denotational semantics for a quite similar language. Our semantics gives rules for type-checking as well as evaluation. Our rules are in fact deterministic and hence can be translated into an interpreter in a conventional programming language such as Pascal.

Related Work

Our work is of course much indebted to that of others. Reynolds, in a pioneering effort, treated the idea of polymorphic types by introducing a special kind of lambda expression (Reynolds, 1974) and McCracken built on this approach (McCracken, 1979). The language Russell introduced dependent types for functions and later for products (Demers and Donahue, 1980). MacQueen and Sethi have done some elegant work on the semantics of a statically typed lambda calculus with dependent types (MacQueen and Sethi, 1982), using the idea that these should be expressed by quantified types: this idea of universally and existentially quantified types was introduced in logic by Girard (Girard, 1972) and used by Martin-Lof (Martin-Lof, 1973) for the constructive logic of mathematics. Mitchell and Plotkin seem to have independently noted the usefulness of existentially quantified types for explaining data abstraction (Plotkin and Mitchell, 1985). We had already noted this utility for dependent products, learning later of the work on Russell and the connection with quantified types. It is a little hard to know who first made these observations; they seem to have been very much "in the air."

A notable difference between our approach and that of others using quantified types is that we take types as values and have have only one kind of lambda expression. Russell also takes types as values, but they are abstract data types with operations, whereas we start with types viewed as simple predicates without operations, building more complex types from this simple basis. The idea of taking bindings as values also appears in (Plotkin, 1981) with a somewhat similar motivation. Our work has been influenced by previous work by one of us with Goguen on the design of the specification language Clear (Burstall and Goguen, 1977).

2. Informal Description of Pebble

This section describes the language, with some brief examples and some motivation. We first go through the conventional features such as expressions, conditionals, and function definitions. Then we present those which have more interest:

- the use of bindings as values with declarations as their types;
- the use of types as values (at compile time);
- the extension of function and product types to dependent types;
- the method of defining polymorphic functions.

Finally we say something about type-checking.

The reader may wish to consult the formal description of values and the formal syntax, given in Section 4, when he is unclear about some point. Likewise the operational semantics, given in Section 5, will clarify exact details of the type-checking and evaluation.

2.1. *Basic Features*

Pebble is based upon lambda calculus with types, using a fairly conventional notation. It is entirely functional and consists of expressions which denote values. This distinction between expressions and values is in accord with our desire to keep our semantics quite close to a practical implementation; for example, we choose to use closures as the values of lambda expressions. Note that in passing from expressions to values we lose type information, e.g., in passing from a binding expression to a binding value or from a lambda expression to a closure.

We start by describing the values, which we write in **this font** for the remainder of this section. They are:

- primitive values: integers and booleans;
- function values: primitive operations, such as +, and closures which are the values of lambda expressions;
- tuples: **nil** and pairs of values, such as **[1, 2]**;
- bindings: values such as **x** ~ **3** which associate a name with a value, sets of these values which associate sets of names and values, and fix bindings which arise in defining recursive functions;
- types

  the primitive types **int** and **bool**
  types formed by × and →
  dependent types formed by ◇ and ▷,
  void, the type whose only element is **nil**

inferred product types formed by ⊗
extended types formed by **xt**
the type **type** which is the type of all types including itself, and
declarations, such as **x: int**, which are the types of bindings;

– symbolic applications: these consist of a function value applied to an argument, written **f! e**. They arise during type-checking. These are not final values of expressions, but are used in the formal semantics.

We now consider the various forms of expressions, putting aside for the moment the details of bindings, declarations, and dependent types, which will be discussed in later sections. These are as follows:

– applications: these are of the form "operator operand," for example, *factorial* 6, with juxtaposition to denote application. Parentheses and brackets are used purely for grouping. If E_1 is an expression of type $t_1 \to t_2$ and E_2 is an expression of type t_1, then $E_1 E_2$ is an expression of type t_2. As an abbreviation we allow infixed operators such as $x + y$ for $+[x, y]$.

– tuples: nil is an expression of type **void**. If E_1 is an expression of type t_1 and E_2 one of type t_2 then $[E_1, E_2]$ is an expression of type $t_1 \times t_2$. The brackets are not significant and may be omitted. The functions fst and snd select components, thus fst[1, 2] is **1**.

– conditionals: IF E_1 THEN E_2 ELSE E_3 where E_1 is of type **bool**.

– local definitions: LET B IN E evaluates E in the environment enriched by the binding B. For example,

$$\text{LET } x\text{: int} \sim y + z \text{ IN } x + \text{abs } x$$

first evaluates $y + z$ and then evaluates $x + \text{abs } x$ with this value for x. The int may be omitted, thus

$$\text{LET } x :\sim y + z \text{ IN...}$$

The binding may be recursive, thus

$$\text{LET REC } f\text{: (int} \to \text{int)} \sim \text{... IN...}$$

We allow E WHERE B as an abbreviation for LET B in E.

– function definitions: functions are denoted by lambda expressions, for example,

$$\lambda x\text{: int} \to \text{int IN } x + \text{abs } x$$

which when applied to 3 evaluates 3 + abs 3, yielding **6**. If T_1 evaluates to t_1, T_2 evaluates to t_2, and E is an expression of type t_2, then

$$\lambda N\text{: } T_1 \to T_2 \text{ IN } E$$

is a function of type $t_1 \to t_2$. The result type T_2 may be omitted. Thus

$$\lambda x\text{: int IN } x + \text{abs } x$$

defines the same function as the previous example. Functions of two or more arguments can be defined by using $\times$, for example,

$$\lambda x\text{: int} \times y\text{: bool} \to \text{int IN...}$$

We allow the abbreviation

$$f(i\text{: int} \to \text{int}) :\sim \ldots$$

for

$$f\text{: (int} \to \text{int)} \sim \lambda i\text{: int} \to \text{int IN...}$$

An example may help to make this all more digestible:

```
LET REC fact(n: int → int) :~
  IF n = 0 THEN 1 ELSE n * fact(n − 1)
IN LET k :~ 2 + 2 + 2 IN
  fact(fst[k, k + 1])
```

This all evaluates to factorial 6. Slightly less dull is

```
LET twice((f: int → int) → (int → int)) :~
  λn: int → int IN f(f n)
IN(twice square)(2)
```

which evaluates to *square*(*square*(2)), that is **16**. We shall see later how we could define a polymorphic version of *twice* which would not be restricted to integer functions.

Note that a lambda expression evaluates to a *closure* which consists of the declaration and body of the lambda expression together with a binding corresponding to the environment in which the lambda expression was evaluated; this gives values for the free variables in the body.

The reader will note the omission of assignment. Its addition would scarcely affect the syntax, but it would complicate the formal semantics by requiring the notion of store. It would also complicate the rules for type-checking, since in order to preserve static type-checking, we would have to make sure that types were constants, not subject to change by assignment. This matter is discussed further in Section 3.5.

2.2. *Bindings and Declarations*

An unconventional feature of Pebble is that it treats bindings, such as $x \sim 3$, as values. They may be passed as arguments and results of functions,

and they may be components of data structures, just like integers or any other values. The expression x: int ~ 3 has as its value the binding $x \sim 3$. A binding is evaluated by evaluating its right hand side and attaching this to the variable. Thus if x is 3 in the current environment, the expression y: int $\sim x + 1$ evaluates to the binding $y \sim 4$. The expression x: int ~ 3 may be written more briefly $x :\sim 3$; the type of 3, which is int, is supplied automatically.

The type of a binding is a declaration. Thus the binding expression $x :\sim 3$ has as its type the declaration x: int. Bindings may be combined by pairing; unlike most other values, a pair of bindings is another binding. Thus $[x :\sim 3, b :\sim \text{true}]$ is also a binding. After LET such a complex binding acts as two bindings "in parallel," binding both x and b. Thus

$$\text{LET } x :\sim 0 \text{ IN LET}[x :\sim 3, y :\sim x] \text{ IN } [x, y]$$

has value [3, 0] not [3, 3], since both bindings in the pair are evaluated in the outer environment. Thus the pair constructor "," is just like any other function. The type of the binding $[x :\sim 3, b :\sim \text{true}]$ is $(x\text{: int}) \times (b\text{: bool})$ since as usual if e_1 has type t_1 and e_2 has type t_2 then $[e_1, e_2]$ has type $t_1 \times t_2$. Using pairing to combine bindings does introduce a left-to-right ordering which is strictly unnecessary, but this representation avoids introducing any extra machinery.

For convenience we have a syntactic sugar for combining bindings "in series." We write this B_1; B_2, which is short for $[B_1, \text{LET } B_1 \text{ IN } B_2]$. There are no other operations on bindings, with the possible exception of equality which could well be provided.

Declarations occur not only as the types of bindings but also in the context of lambda expressions. Thus in

$$\lambda x\text{: int} \to \text{int IN } x + 1$$

x: int is a declaration, and hence x: int $\to$ int is a type. In fact you may write any expression after the λ provided that it evaluates to a type of the form $d \to t$ where d is a declaration. To make two argument lambda expressions we simply use a $\times$ declaration, thus

$$\lambda x\text{: int} \times y\text{: int} \to \text{int IN } x + y$$

which is of type int $\times$ int $\to$ int, and could take [2, 3] as an argument. This introduces a certain uniformity and flexibility into the syntax of lambda expressions.

We may write some unconventional expressions using bindings as values. For example,

LET $b :\sim (x :\sim 3)$ IN LET b IN x

which evaluates to 3. Another example is

LET $f(b\colon (x\colon \text{int} \times y\colon \text{int}) \to \text{int}) :\sim$ LET b IN $x + y$
IN $f[x :\sim 1, y :\sim 2]$

which also evaluates to 3. Here f takes as argument not a pair of integers but a binding.

The main intended application of bindings as values is in elucidating the concept of a parameterised module. Such a module delivers a binding as its result; thus, a parameterised module is a *function* from bindings to bindings. Consider a module which implements sorting, requires as parameter a function *lesseq* on integers, and produces as its result functions *issorted* and *sort*. It could be represented by a function from bindings whose type would be

lesseq: (int × int → bool) →
(*issorted*: (list int → bool) × *sort*: (list int → list int))

We go into this in more detail in Section 3.1.

If a module requires as its parameter a binding, say one binding to f and h, it does no harm to give it a bigger but compatible one binding to f and g and h. This is often called "inheritance" or "subclassing." So when we apply a function to an argument which is a binding, a coercion is done on this binding to "shrink" it down to the right shape, this shape being determined by the declaration of the parameter of the function. For example we accept

LET *Arith* :~ (*lesseq*: (int × int → bool) ~ ..., *add*: (int × int → int) ~ ...) IN
LET *SortModule*(*lesseq*: (int × int → bool) → (*issorted*:... × *sort*:...) :~ ... IN
SortModule(*Arith*)

in which *SortModule* needs *lesseq* and it gets *lesseq* and *add*.

Since " ~ " is a function, it also coerces a binding; thus we accept

b: (*lesseq*:...) ~ (*lesseq*:... ~ ..., *add*:... ~ ...)

Pebble also has an anti-LET, which impoverishes the environment instead of enriching it:

IMPORT B IN E

evaluates E in an environment which contains *only* the bindings in B, for example,

IMPORT B IN x

The value of this expression is the value of x in the binding B, if x is indeed bound by B. Otherwise it has no value. This is very useful if B is a named collection of values from which we want to obtain the one named x. If we write simply, LET B IN x, and x is missing from B, we would pick up any

x that happens to be in the current environment. The construction in the example is so useful that we provide the syntactic sugar $B \$ x$ for it. Thus *stack* \$ *pop* is the value of *pop* in the binding *stack*.

2.3. *Types*

We can now explain how Pebble handles types. It may be helpful to begin by discriminating between some of the different senses in which the word "type" is customarily used. We use ADT to abbreviate "abstract data type."

– Predicate type, simply denoting a set of values. Example: bool considered as {true, false}.

– Simple ADT, a single predicate type with a collection of associated operations. Example: *stack* with particular operations:

$$push: (\text{int} \times stack \to stack) \sim \ldots, \text{ etc.}$$

– Multiple ADT, several predicates (zero or more) with a collection of associated operations. Example: *point* and *line* with particular operations:

$$intersection: (line \times line \to point) \sim \ldots, \text{ etc.}$$

– ADT declaration, several predicate names with a collection of associated operation names, each having inputs and outputs of given predicate names. Example: predicate names *point* and *line* with operator names:

$$intersection: (line \times line \to point), \text{ etc.}$$

The simple ADT is a special case of the multiple ADT which offers notational and other conveniences to language designers. For the ADT declaration we may think of a collection of (predicate) type and procedure declarations, as opposed to the representations of the types and the code for the operations.

Some examples of how these concepts appear in different languages may help. The last column in Table I gives the terminology for many sorted algebras.

In Pebble we take as our notion of type the first of these, predicate types. Thus a type is simply a means of classifying values. We are then able to define entities which are simple ADTs, multiple ADTs, and ADT declarations. To do this we make use of the notions of binding and declaration already explained, and the notion of dependent type explained below.

There are two methods of achieving "abstraction" or "hiding" of data type implementations. One method is by parameterisation. If a module

TABLE I

	Pascal	CLU	Mesa	Ada	Russell	ML	SML	Algebra
Predicate	Type	Type	Type	Type	—	Type	Type	Sort
Simple ADT	—	Cluster	—	—	Type	—	—	Algebra
Multiple ADT	—	—	Implementation	Package body	—	Abstract Type	Structure	Algebra
ADT declaration	—	—	Interface	Package spec	—	—	Signature	Signature

takes an ADT as a parameter, when writing the body of the module the parameter has access to the ADT declaration which describes this parameter but not to the ADT itself. The other method is to use a "password," chosen by the programmer of the ADT or uniquely generated automatically, to protect values of the abstract type. These approaches are illustrated in Pebble in Sections 3.1 and 3.2.

Pebble treats types as values, just like integers and other traditional values. We remove the sharp distinction between "compile time" and "run time," allowing evaluation (possibly symbolic) at compile time. This seems appropriate, given that one of our main concerns is to express the linking of modules and the checking of their interfaces in the language itself. Treating types as values enriches the language to a degree at which we might lose control of the phenomena, but we have adopted this approach to get a language which can describe the facilities we find in existing languages such as Mesa and Cedar. A similar but more conservative approach, which maintained the traditional distinction between types and values, was pursued by David MacQueen at Bell Labs, with some collaboration by one of us (R.B.). He has recently applied these ideas to the design of a module facility for ML (MacQueen, 1984); this is incorporated in Standard ML (SML) (Harper, MacQueen, and Milner, 1986). The theoretical basis for this work has been developed in (MacQueen and Sethi, 1982; MacQueen, Plotkin, and Sethi, 1984).

Allowing "type" to be a type causes inconsistency in logic systems which use the "propositions as types" idea, as shown by Girard (1972). However for programming languages inconsistency does not arise and a denotational semantics can be given using closures, a form of retract (Amadio and Longo, 1986). The language could be reformulated if desired using a type hierarchy, but at a cost in complication.

2.4. *Dependent Function Types and Polymorphism*

A function is said to be polymorphic if it can accept an argument of more than one type; for example, an equality function might be willing to accept either a pair of integers or a pair of booleans. To clarify the way Pebble handles polymorphism we should first discuss some different phenomena which may be described by this term. We start with a distinction (due we believe to C. Strachey) between *ad hoc* and *universal* polymorphism.

- Ad hoc polymorphism: the code executed depends on the type of the argument; e.g., "print 3" involves different code from "print 'nonsense'".

- Universal polymorphism: the same code is executed regardless of the type of the argument, since the different types of data have uniform

representation, e.g., *reverse*[1, 2, 3, 4] and *reverse*[true, false, false]. (We write [...] for lists in examples.)

We have made this distinction in terms of program execution, lacking a mathematical theory. Recently Reynolds has offered a mathematical treatment (Reynolds, 1983).

In Pebble we take universal polymorphism as the primitive idea. We are able to program ad hoc polymorphic functions on this basis (see Section 3.3 on generic types). But universal polymorphism may itself be handled in two ways: *explicit parameterisation* or *type inferences*.

– Explicit parameterisation: when we apply the polymorphic function we pass an extra argument (parameter), namely the type required to determine the particular instance of the polymorphic function being used. For example, *reverse* would take an argument t which is a type, as well as a list. If we want to apply it to a list of integers we would supply the type int as the value of t, writing *reverse*(int)[1, 2, 3, 4] and *reverse*(bool) [true, false, false]. To understand the type of *reverse* we need the notion of dependent type, to be introduced later. This approach is due to Reynolds (1974) and is used in Russell and CLU.

– Type inference: the type required to instantiate the polymorphic function when it is applied to a particular argument need not be supplied as a parameter. The type-checker is able to infer it by inspecting the type of the argument and the type of the required result. A convenient and general method of doing this is by using unification on the type expression concerned (Milner, 1978); this method is used in ML (Gordon, Milner, and Wadsworth, 1979). For example, we may write *reverse*[1, 2, 3, 4]. Following Girard (1972) we may regard these type variables as universally quantified. The type of *reverse* would then be

$$\text{for all } t\text{: type.list}(t) \rightarrow \text{list}(t).$$

This form is used by MacQueen and Sethi (1982).

In Pebble we adopt the explicit parameterisation form of universal polymorphism. This has been traditional when considering instantiation of modules, as in CLU or Ada generic types. To instantiate a module we must explicitly supply the parameter types and procedures. Thus before we can use a generic Ada package to do list processing on lists of integers, we must instantiate it to integers. The pleasures of type inference polymorphism as in ML seem harder to achieve at the module level; in fact one seems to get involved with second order unification. This is an open area for research. It must be said that explicit parameterisation makes programming in the kernel language more tedious. However, Section 2.6 describes sugar which

automatically supplies a value for the type parameter when a function is applied, at the cost of some extra writing when it is defined.

For example, we might want to define a polymorphic function for reversing a pair, applied thus

swap[int, bool][3, true],

which evaluates to [true, 3]. Here *swap* is applied to the pair of types [int, bool] and delivers a function whose type is int × bool → bool × int. The type of *swap* is what we will call a *dependent type* (Girard, 1972; Demers and Donahue, 1980). (A mild abuse of language, since it is really the result of applying *swap* to [int, bool] which is dependent, rather than the type of *swap* itself.) We will need two kinds of dependent type constructor, one analogous to → for dealing with functions, the other analogous to × for dealing with pairs. We consider the former here, and deal with the latter in the next section.

We might think naively that the type of *swap* would be

$$(\text{type} \times \text{type}) \rightarrow (t_1 \times t_2 \rightarrow t_2 \times t_1)$$

but of course this is nonsense because the type variables t_1 and t_2 are not bound anywhere. The fact is that *the type of the result depends on the values of the arguments*. Here the arguments are a pair of types and t_1 and t_2 are the names for these values. We need a special arrow →» instead of → to indicate that we have a dependent type; to the left of the →» we must declare the variables t_1 and t_2. So the type of *swap* is actually

$$(t_1\text{: type} \times t_2\text{: type}) \twoheadrightarrow (t_1 \times t_2 \rightarrow t_2 \times t_1).$$

In order to have lambda abstraction be the only name-binding mechanism, we introduce an operation ▷ and take this as syntactic sugar for

$(t_1$: type × t_2: type) ▷ λ *B*: (t_1: type × t_2: type) → type IN
  LET *B* IN ($t_1 \times t_2 \rightarrow t_2 \times t_1$)

which evaluates to

(t_1: type × t_2: type) ▷ c

where c is the closure which is the value of the λ expression after the ▷. Thus ▷ is a new value constructor for dependent function types. For example, the type of *swap*[int, bool] is int × bool → bool × int.

We may now define *swap* by

swap(t_1: type × t_2: type) →» ($t_1 \times t_2 \rightarrow t_2 \times t_1$) :~
  λx_1: t_1 × x_2: t_2 → $t_2 \times t_1$ IN [x_2, x_1]

Another example would be the list reversing function

REC *reverse*(*t*: type →» (list *t* → list *t*)) :~
  λ*l*: list *t* → list *t* IN
    IF *l* = nil THEN *l* ELSE *append*(*reverse tail l*, *cons*(*head l*, nil))

2.5. *Dependent Product Types*

A similar phenomenon occurs with the type of pairs. Suppose for example that the first element of a pair is to be a type and the second element is to be a value of that type; for example [int, 3] and [bool, false]. The type of all such pairs may be written (*t*: type) ×× *t*. As we did with →», we take its value to be *t*: type ◇ *c* where *c* is the closure which is the value of λ*t*: type → type IN *t*, and ◇ is a new value constructor for dependent product types. It is a dependent type because *the type of the second element depends on the value of the first.* Actually it is more convenient technically to let this type include all pairs whose first element is not just a type but a binding of a type to *t*. So expressions of type (*t*: type) ×× *t* are [*t* :~ int, 3] and [*t* :~ bool, false] for example.

A more realistic example might be

Automaton: type ~ (*input*: type × *state*: type × *output*: type)
  ×× (*tf*: (*input* × *state* → *state*) × *of*: (*state* → *output*))

Values of the type *Automaton* are pairs, consisting of

(i) three types called *input*, *state*, and *output*;

(ii) a transition function, *tf*, and an output function, *of*.

By "three types called *input*, *state*, and *output*" we mean a binding of types to these names. Section 3 illustrates various ways of using dependent product types to describe modules.

The simplest use of dependent products is illustrated by *Automaton*, in which the first of the product is a type. We can also use dependent products to provide union types. (Indeed what we call "dependent product" is often called "disjoint union of a family of types" by logicians.) When we use *Automaton*, we are not concerned with what the types *input*, *state*, and *output* might be, but only with how the functions *tf* and *of* transform values of these types. Sometimes, however, we may wish to test the value of fst *x* and take advantage of what this tells us about the type of snd *x*. For example, consider

t: type ~ (*tag*: bool ×× (IF *tag* THEN int ELSE real)).

If *x* has type *t*, then if *x* $ *tag* = true, snd *x* has type int. Often *t* is called a *union* or *sum* type and written int ⊕ bool. The expressions (true, 3) and

(false, 3.14) have type t, so that the separate injection functions commonly provided for making union values are not needed.

We might try to write

λx: t IN IF x \$ *tag* THEN snd $x + 1$ ELSE floor (snd x)

but this will not type-check, because the Pebble type-checker is unable to keep track of the fact that after THEN the value of x \$ *tag* is true. (To do so would be a major extension of the notion of type-checking.) We therefore introduce an AS construct which can be used like this:

λx: t IN IF x \$ *tag* THEN (x AS true) + 1 ELSE floor(x AS false).

In general, if E has type $d_1 \times\times t_2$, and fst $E = E_1$, then E AS E_1 has type (LET $d_1 \sim E_1$ IN t_2) and value snd E. However, if fst $E \neq E_1$, then E AS E_1 is undefined, and hence the value of any expression in which this happens is undefined. It is the programmer's responsibility to establish the precondition (fst $E = E_1$) before any occurrence of E AS E_1. For the future we expect to add exceptions to Pebble, and then a failing AS expression will have an exception as its value instead of being undefined. (The AS device is something of a patch, and we have since investigated some alternatives.)

Using this primitive, various kinds of sugar for unions can be devised. As one example, we offer the following:

N_1: $t_1 \oplus \ldots \oplus N_j$: t_j for *tag*: string $\times\times$ (IF *tag* = "N_1" THEN t_1 ELSE...
ELSE IF *tag* = "N_j" THEN t_j ELSE void)

and

CASE $N :\sim E$ OF
  N_1 THEN E_1
| ...
| N_k THEN E_k
ELSE E_0

for

IF E \$ *tag* = "N_1" THEN LET $N :\sim (E$ AS "N_1") IN E_1
ELSE ... ELSE
IF E \$ *tag* = "N_k" THEN LET $N :\sim (E$ AS "N_k") IN E_k
ELSE E_0

For example, if

$T :\sim$ (*one*: int $\oplus$ *two*: int $\times$ int $\oplus$ *many*: list int);

then we can write z: $T \sim$ ("two", (5, 10))), and the function

sum(y: $T \to$ int) :~
  CASE x :~ y OF
    one THEN x
  | *two* THEN fst x + snd x
  | *many* THEN IF x = nil THEN 0 ELSE head x + *sum*("many", tail x)
    ELSE error ( )

will add up the integers in its argument, so that *sum*(z) evaluates to 15.

This sugar is somewhat arbitrary, perhaps reflecting the fact that a satisfactory syntax for discriminating the cases of a union type has yet to be devised in any programming language.

2.6. *Polymorphism without Tears*

Although we are able to define polymorphic functions like *swap* or *reverse*, it is irritating that we must supply an explicit type argument at each call of the function. Why can't we say *swap*[3, true] instead of *swap*[int, bool][3, true]? ML accomplishes this by using unification to infer the int and bool from the type of [3, true].

We propose that the Pebble programmer, deprived of unification, should at least be allowed to supply a function which calculates these parameter types from the type of the actual argument.

Consider the list reversing function, which we defined thus with parameter type t,

REC *reverse*(t: type) →» (list $t \to$ list t) :~
  λl: list $t \to$ list t IN IF l = nil THEN... ELSE...

For our purposes we prefer the "uncurried" version which takes two arguments

REC *reverse*(((t: type ×× l: list t) →» list t) :~
  IF l = nil THEN... ELSE...

which is used thus *reverse* (int, [1, 2, 3])—we write [...] for lists in examples. We would like to write *reverse'*[1, 2, 3]. So in general *reverse' E* should mean the same as *reverse*(t, E), where t is a type obtained by inspecting the type of E. If we write τE for the type of E, t should actually be list$^{-1}(\tau E)$, where list^{-1} is the inverse of the type constructor list. (We allow ourselves to write list^{-1} for the lengthy name listinverse.) So *reverse'*[1, 2, 3]) means the same as *reverse*(list^{-1}(list int), [1, 2, 3]), i.e., *reverse*(int, [1, 2, 3]). We shall call list^{-1} the "discovery function." It discovers the appropriate type parameter for polymorphic *reverse* by looking at the type of the actual argument. Our idea is that the programmer should supply the discovery function as part of the type of *reverse'*; then by look-

ing at the type of *reverse'* we can coerce the argument [1, 2, 3] to (int, [1, 2, 3]).

An alternative approach would be to not demand a discovery function but instead use a general matcher, just as unification is used in ML. Then t would get bound to int by matching list t against list int. Since we have not just types but functions from types to types we fear second order complications in such a matcher and stick with discovery functions.

To put the discovery function into the type we need a new type constructor "⊗" called "inferred product." We write

REC *reverse'* ~ λ (t: type ×× l: list t) ⊗ list^{-1} →→ t
IN IF... THEN... ELSE...

(very like *reverse* but with "⊗ list^{-1}" inserted). When we write *reverse'*[1, 2, 3] the type-checker finds the type of [1, 2, 3], namely list int, applies the discovery function to it, binds (t: type ×× l: list t) to (int, [1, 2, 3]), and then proceeds to evaluate the type of the result; in due course the body is evaluated.

A more elaborate example is

compose((t_1: type × t_2: type × t_3: type ×× f_1: (t_1 → t_2) × f_2: (t_2 → t_3))
⊗ (λT: type IN(→$^{-1}$(fstt T), snd(→$^{-1}$(sndt T)))) →→ (t_1 → t_3)) :~
λx: t_1 IN f_2(f_1 x)

Here we have used functions fstt and sndt which extract the first and second parts of a cross type to decompose the argument type.

An important property of the discovery function coercion is that it does not endanger the security of the type system. When [1, 2, 3], with type list int, is coerced to the type t: type ×× l: list t ⊗ list^{-1}, the discovery function list^{-1} is applied to list int to yield int. This is only a guess about the type that is needed. The guess is paired with the original expression to give (int, [1, 2, 3]), and this expression must have type t: type ×× l: list t. The ⊗ constructor and the discovery function play no role in this type-check, and if the type int guessed by the discovery function is wrong, the type-check will fail. In fact, the value of the expression [1, 2, 3] plays no role either; only its type is important, since the coercion is a function from the type list int to the type t: type ×× l: list t, namely

λ N': list int → (t: type ×× l: list t)
IN(list^{-1}(list int), N')

which type-checks because list^{-1}(list int) = int and (int, N') has type (t: type ×× l: list t) when N' has type list int.

2.7. *Extended types*

The inferred product allows us to compute the type argument of an application when the type involved is the result of a type constructor such

as list or →. Often, however, we want to deal with *abstract types*. For example, consider the declaration

List: type ~ (*t*: type ×× *elem*: type ××
empty: *t* × *cons*: (*elem* × *t* → *t*) ×
head: (*t* → *elem*) × *tail*: (*t* → *t*))

We can write a different *reverse*, which works on these abstract lists:

reverse(*L*: *List* ×× *l*: *L* $ *t* →» *L* $ *t*): ~ IF *l* = *L* $ *empty* THEN *l* ELSE...

This *reverse* uses *L* $ *empty*, *L* $ *cons*, *L* $ *head*, and *L* $ *tail* to manipulate values of type *L* $ *t*, and it works quite independently of what particular type *L* $ *t* happens to be, i.e., independently of the representation of lists. In Sections 3.2–3.3 this style of programming is discussed further.

If we have *LL*: *List* with *LL* $ *elem* = int, then

LL $ *cons*(1, *LL* $ *cons*(2, *LL* $ *cons*(3, *LL* $ *empty*)))

has type *LL* $ *t*; we shall write it *LL*[1, 2, 3], to emphasize the similarity with the previous case. As before, we would like to write *reverse'* *LL*[1, 2, 3] rather than *reverse*(*LL*, *LL*[1, 2, 3]), but the situation is more complicated since *LL* is not a type. In fact, the type of this *reverse* is *L*: *List* ×× *L* $ *t* →» *L* $ *t*, as we saw earlier. If we want to coerce *LL*[1, 2, 3] into (*LL*, *LL*[1, 2, 3]), we have nothing to go on except the type of *LL*[1, 2, 3]. We therefore must find some way to incorporate *LL* in the type of *LL*[1, 2, 3], so we can write a discovery function that will extract it.

To accomplish this, we introduce the notion of an *extended type*; such a type can be derived from a binding whose first component is *N*: *t* for some type *t*. The primitive xt converts a binding and its type (a declaration) to an extended type; thus

LX :~ xt(*List*, *LL*)

defines a type. We make the *equivalence* rule that if an expression *E* has the type *LL* $ *t* (the base type), then it also has the type xt(*List*, *LL*) (the extended type), and vice versa. In other words, the xt constructor attaches some values to the type *LL* $ *t*, but it does not change the predicate which determines whether an expression has that type.

We want *LX* to be the principal type of *LL*[1, 2, 3]. Then we can define

REC *reverse'*(*L*: *List* ×× *l*: xt(*List*, *L*) ⊗ xtd^{-1}(*List*) →» xt(*List*, *L*)) :~ ...

and write *reverse'*(*LL*[1, 2, 3]), obtaining another value of type *LX*. Here we have used the inverse constructor xtd^{-1}(*List*), with type (type → *List*); when applied to xt(*List*, *LL*) it yields *LL*. The xtd^{-1} function is a convenient specialization of xt^{-1}, which maps xt(*List*, *LL*) into the pair

(*List*, *LL*), just as $\times^{-1}$ maps $t_1 \times t_2$ into the pair (t_1, t_2). For the definition of xtd^{-1} in terms of xt^{-1}, see below or Table VII.

It is convenient to introduce a coercion from bindings such as *LL* to extended types such as *LX*, turning *LL* into xt(*List*, *LL*). With this we never have to write xt explicitly, but can just say

REC *reverse'*(*L*: *List* ×× *l*: $L \otimes \text{xtd}^{-1}(List) \twoheadrightarrow L$) :~ ...

By adding another coercion, from xt(*List*, *LL*) to (*L*: *List*) ×× *l*: *L*, we can simplify this further to

REC *reverse'*(*L*: *List* ×× *l*: $L \twoheadrightarrow L$) :~ ...

The second coercion is just a specialization of the ⊗ coercion to the discovery function $\text{xtd}^{-1}(List)$. More generally, it coerces an expression *E* of type xt(*b*, *d*) to $((\text{xtd}^{-1}\ d)\ \text{xt}(b, d), E)$, or simply (*b*, *E*). These two coercions are described precisely in lines (h8-9) of the coerceF rule in Table VI.

There is one subtlety in type-checking expressions involving extended types. The result type of *LL* \$ *cons* is *LL* \$ *t*, not *LX*. Since by the equivalence rule for extended and base types, any expression with one of these types also has the other type, at first sight this causes no trouble. But suppose we write *reverse'*(*LL*[1, 2, 3]). This is short for *reverse'* (*LL* \$ *cons*(1, ...)) This expression does not type-check, because the argument of *reverse'* must have a *principal* type of the form xt(*List*, *L*) on which $\text{xtd}^{-1}(List)$ can work to extract *L*. But the principal type of *LL* \$ *cons*(1, ...) is *LL* \$ *t*, which does not have this form; it needs to be *LX*, which does.

To solve this problem we introduce a primitive, written as a postfix ↑, which elevates *LL* into a binding with the same value but a different type. The type we want is

List' :~ *t*: type ×× *elem*: type ×× *empty*: *LX* × *cons*: (*LX* × *elem* → *LX*) × ...

This type is obtained from *List* by substituting *LX*, which is xt(*List*, *LL*), for *t* after the first ××. The ↑ primitive is defined precisely in Table VI. Now we can write *LL*↑ \$ *cons*(1, ...), which has type *LX*, as desired. We have

REC *reverse'*(*L*: *List* ×× *l*: $L \twoheadrightarrow L$) :~
  IF *l* = *L*↑ \$ *empty* THEN *l*
  ELSE *append*(*reverse L*↑ \$ *tail*(*l*)), *L*↑ [*L*↑ \$ *head*(*l*)])

A very common situation in an abstract type is to have many functions that take a value of the abstract type as their first argument. For instance *head* and *tail* are such functions for the *List* abstraction, *push* and *pop* for the *Stack* abstraction, and so forth. If *l* has type *LL* (actually xt(*List*, *LL*), we can write *LL*↑ \$ *head*(*l*) to apply the proper *head* function. An attrac-

tive sugar for this is *l.head*. In general, we write $E.N$ for $b\uparrow \$\, N(E)$ if E has type $\mathrm{xt}(d, b)$. To handle additional arguments, this is extended to write $E_1.N(E_2)$ for $b\uparrow \$\, N(E_1, E_2)$, for example, s.push(3) if s is a *Stack* with the obvious *push*: $(t \times \mathrm{int} \to t)$. The programmer can think of *push* as an operation on s without worrying about just which abstraction is supplying it. The power of this notation has been demonstrated by Simula and Smalltalk.

We can now write a final *reverse'* using the dot notation

REC *reverse'*(*L*: *List* ×× *l*: $L \twoheadrightarrow L$) :~
IF $l = L\uparrow \$\, empty$ THEN *l* ELSE *append*(*reverse l.tail*, $L\uparrow$ [*l.head*])

Now we have a neat function which works for any representation of lists.

2.8. *Type-Checking*

Given an expression in Pebble, we first type-check it and then evaluate it. However, the type-checking will involve some evaluation; for example, we will have to evaluate subexpressions which denote types and those which make bindings to type variables. Thus there are two distinct phases of evaluation: evaluation during type-checking and evaluation proper to get the result value. These both follow the same rules, but evaluation during type-checking may make use of symbolic values at times when the actual values are not available; this happens when we type-check a lambda expression.

For each form of expression we need

(i) a type-checking rule with a conclusion of the form: E has type t.

(ii) an evaluation rule with a conclusion of the form: E has value e.

The type-checking rule may evoke the evaluation rules on subexpressions, but the evaluation rule should not need to invoke type-checking rules.

For example, an expression of the form LET... IN... is type-checked using the following rules.

The type of LET B IN E is found thus:

If the type of B is void then it is just the type of E.

If the type of B is N: t_0 then it is the type of E in a new environment computed thus: evaluate B and let e_0 be the right hand side of its value; the new environment is the old one with N taking type t_0 and value e_0.

If the type of B is $d_1 \times d_2$ then evaluate B and let b_2 be the second of its value; now the result is the type of LET fst B IN LET b_2 IN E.

If the type of B is a dependent type of the form $d_1 \diamond f$ then this must be reduced to the previous $d_1 \times d_2$ case by applying f to the binding fst B to get d_2.

The type of a binding of the form $D \sim E$ is the value of D if it is void and E has type void, or if it is N: t and E has type t, or if it is $d_1 \times d_2$ and $[d_1 \sim \text{fst } E,\ d_2 \sim \text{snd } E]$ has type $d_1 \times d_2$; otherwise, if the value of D is a dependent type of the form $d_1 \diamond f$, then this must be reduced to the $d_1 \times d_2$ case by applying f to the binding $(d_1 \sim \text{fst } E)$ to get d_2.

Note that when we write $d_1 \sim \text{fst } E$ we mean strictly the expression corresponding to d_1 rather than the value d_1.

The type of a recursive binding REC $D \sim E$ is just the value of D, provided that a check on the type of E succeeds.

The type of a binding which is a pair is calculated as usual for a pair of expressions.

The value of a binding of the form $D \sim E$ is as follows:

If the value of D is void then nil.

If the value of D is N: t then $N \sim e$ where e is the value of E.

If the value of D is $d_1 \times d_2$ then the value of $(d_1 \sim \text{fst } E,\ d_2 \sim \text{snd } E)$.

If the value of D is a dependent type then we need to reduce it to the previous case (as before).

A couple of examples may make this clearer. We give them as informal proofs. The proofs are not taken down to the lowest level of detail, but display the action of the rules just given.

EXAMPLE.

LET x: $(\text{int} \times \text{int}) \sim [1 + 1, 0]$ IN fst x

has type int (and value 2). To show this, we first compute the type of the binding: x: $(\text{int} \times \text{int}) \sim [1 + 1, 0]$ has type x: $(\text{int} \times \text{int})$ because x: $(\text{int} \times \text{int})$ has type type and x: $(\text{int} \times \text{int})$ has value x: $(\text{int} \times \text{int})$ and $[1 + 1, 0]$ has type $\text{int} \times \text{int}$.

This is of the form N: t, so we evaluate the binding,

$$x\text{: } (\text{int} \times \text{int}) \sim [1 + 1, 0] \text{ has value } x \sim [2, 0].$$

We type-check fst x in the new environment formed by adding

$$[x\text{: } (\text{int} \times \text{int})] \text{ and } [x \sim [2, 0]].$$

In this environment fst x has type int. This is the type of the whole expression.

Here is a second rather similar example, in which LET introduces a type name. It shows why it is necessary to evaluate the binding after the LET, not just type-check it. We need the appropriate binding for any type names

which may appear in the expression after IN. Here t in t: type $\sim$ int is such a name, and we need its binding to evaluate the rest of the expression.

EXAMPLE.

LET t: type $\sim$ int IN
  LET x: $t \sim 1$ IN $x + 1$

has type int (and incidentally value 2). We first type-check the binding of the first LET,

$$t\text{: type} \sim \text{int has type } t\text{: type and value } t \sim \text{int}$$

In the new environment formed by adding [t: type] and [$t \sim$ int] we must type-check LET x: $t \sim 1$ IN $x + 1$. This has type int because

x: $t \sim 1$ has type x: int and
x: $t \sim 1$ has value $x \sim 1$ and

in the new environment formed by adding [x: int] and [$x \sim 1$], $x + 1$ has type int.

What about type-checking lambda expressions? For expressions such as

λx: int $\rightarrow$ int IN $x + 1$

this is straightforward. We can simply type-check $x + 1$ in an environment enriched by [x: int]. But we must also consider polymorphic functions such as

λt: type $\rightarrow\!\!\rightarrow$ $(t \rightarrow t)$ IN λx: $t \rightarrow t$ IN E

We would like to know the type of x when type-checking the body E, but this depends on the argument supplied for t. However, we want the lambda expression to type-check no matter what argument is supplied, since we want it to be universally polymorphic. Otherwise we would have to type-check it anew each time it is given an argument, and this would be dynamic rather than static type-checking. So we supply a dummy, symbolic value for t and use this while type-checking the rest of the expression. That is, we type-check

λx: $t \rightarrow t$ IN E

in an environment enriched by [t: type] and [$t \sim$ newconstant], where newconstant is a symbolic value of type type, distinct from all other symbolic values which may occur in this environment. This distinctness is

ensured by keeping a depth counter in the environment and using it to construct newconstant. Under this regime a function such as

λt: type →» $(t \to t)$ IN $(\lambda x$: $t \to t$ IN $x)$

will type-check (it has type denoted by t: type →» $(t \to t)$) but

λt: type →» $(t \to t)$ IN $(\lambda x$: $t \to t$ IN $x + 1)$

will fail to type-check because it makes sense only if t is int.

Thus it is necessary that at type-checking time evaluation can give a symbolic result, since we may come across newconstant. How do we apply a function to such a value? We introduce a value constructing operator ! to permit the application of a function to a symbolic argument. So if e is symbolic the result of applying f to e is just $f!e$. Similarly, if f is symbolic the result of applying f to e is just $f!e$. This enables us to do symbolic evaluation at compile time and to compare types as symbolic values.

There are no operations on types except the constructors and their inverses. Thus there is no way to compute an integer, say, from a type. Assuming that the result of a run-time computation is an integer or boolean, or structure thereof, rather than a type, there is no need to carry types around at run time, and, for example, the pair [int, 3] can be represented at run time simply by 3. (An exception is "extended types" (Section 2.6) which are bindings acting as types.) Thus in Pebble, types act as values at compile time, but although we may formally think of them as values at run time they play no computational role.

Since our language has dependent types, an expression can have more than one type. For example $(t :\sim \text{int}, 3)$ has type t: type × int, but it also has the dependent type t: type ×× t. The former, called the "principal type," is calculated by the type-checking algorithm. To type-check $(\lambda(t$: type ×× $t) \to \ldots$ IN...$)(t :\sim \text{int}, 3)$ we need an algorithm to verify that $(t :\sim \text{int}, 3)$ has type t: type ×× t.

We must admit that, although our type-checker is precisely defined by our operational semantics, we have no good mathematical characterisation of when it will succeed. We could have made it weaker and probably easier to characterise, by restricting the amount of symbolic evaluation carried out at compile time, but this would not necessarily help the programmer. We would welcome suggestions for characterisation.

3. Applications

This section presents a number of applications of Pebble, mainly to programming in the large: interfaces and implementations, and abstract data types. We also give treatments of generic types, union types, recursive types such as list, and assignment. The point is to see how all these facilities can be provided simply in Pebble.

3.1. *Interfaces and Implementations*

The most important recent development in programming languages is the introduction of an explicit notion of *interface* to stand between the implementation of an abstraction and its clients. To paraphrase Parnas:

– An interface is the set of assumptions that a programmer needs to make about another program in order to show the correctness of his program.

Sometimes an interface is called a *specification* (e.g., in Ada, where the term is *package specification*). We will call the other program an *implementation* of the interface, and the program which depends on the interface the *client*.

In a practical present day language, it is not possible to check automatically that the interface assumptions are strong enough to make the client program correct, or that an implementation actually satisfies the assumptions. In fact, existing languages cannot even express all the assumptions that may be needed. They are confined to specifying the names and types of the procedures and other values in the interface.

This is exactly the function of a definition module in Mesa or Modula2, a package specification in Ada, or a module type in Euclid. These names and types are the assumptions which the client may make, and which the implementation must satisfy by providing values of the proper types. In one of these languages we might define an interface for a real number abstraction as follows:

```
interface Real;
  type real;
  function plus(x: real; y: real): real;
  ...
end
```

and an implementation of this interface, using an existing type *float*, might look like this:

```
implementation RealFl implements Real;
  type real = float;
  function plus(x: real; y: real): real;
    begin
      if ... then ... else ... end;
      return ...;
    end;
  ...
  end
```

In Pebble an interface such as *Real* is simply a declaration for a type

Real $ *real* and various functions such as *plus*; an implementation of *Real* is a binding whose type is *Real*. Here is the interface:

Real: type ~ (*real*: type ××
plus: (*real* × *real* → *real*) × ...);

Note that this is a dependent type: the type of *Real* $ *plus* depends on the value of *Real* $ *real*.

Now for the implementation, a binding with type *Real*. It gives *real* the value *float*, which must denote some already-existing type, and it has an explicit λ-expression for *plus*.

RealFl: *Real* ~ [*real* :~ *float*;
plus :~ λ*x*: *real* × *y*: *real* → *real* IN
(IF ... THEN ... ELSE ...), ...]

On this foundation we can define another interface *Complex*, with a declaration for a *mod* function which takes a *Complex* $ *complex* to a *RealFl* $ *real*,

Complex: type ~ (*complex*: type ××
mod: *complex* → *RealFl* $ *real* × ...)

If we do not wish to commit ourselves to the *RealFl* implementation, we can define a parameterised interface *MakeComplex*, which takes a *Real* parameter:

MakeComplex(*R*: *Real* → type) :~ (*complex*: type ××
mod: *complex* → *R* $ *real* × ...)

Then the previous *Complex* can be defined by

Complex: type ~ *MakeComplex*(*RealFl*)

This illustrates the point that a module is usually a function producing some declaration or binding (the one it defines) from other declarations and bindings (the interfaces and implementations it depends on).

Now the familiar cartesian and polar implementations of complex numbers can be defined, still with a *Real* parameter. This is possible because the implementations depend on real numbers only through the elements of a binding with type *Real*: the *real* type, the *plus* function, etc.

MakeCartesian(*R*: *Real* →» *MakeComplex*(*R*)) :~
[*complex* :~ *R* $ *real* × *R* $ *real*;
mod :~ λ*c*: *complex* → *R* $ *real*
IN *R* $ *sqrt*((fst *c*)² + (snd *c*)²), ...];

MakePolar(*R*: *Real* →» *MakeComplex*(*R*)) :~
[*complex* :~ *R* $ *real* × *R* $ *real*;
mod :~ λ*c*: *complex* → *R* $ *real* IN fst *c*,...];

These are functions which, given an implementation of *Real*, will yield an implementation of *MakeComplex*(*Real*). To get actual implementations of *Complex* (which is *MakeComplex*(*RealFl*)), we apply these functions:

Cartesian: *Complex* ~ *MakeCartesian*(*RealFl*)
Polar: *Complex* ~ *MakePolar*(*RealFl*);

If we do not need the flexibility of different kinds of complex number, we can dispense with the *Make* functions and simply write

Cartesian: *Complex* ~ [*complex* :~ $R \times R$;
mod :~ λc: *complex* → *R* IN
RealFl $ *sqrt*((fst *c*)^2 + (snd *c*)^2), ...],
Polar: *Complex* ~ [*complex* :~ $R \times R$;
mod :~ λc: *complex* → *R* IN fst *c*, ...]
WHERE *R*:~ *RealFl* $ *real*

To show how far this can be pushed, we define an interface *Transform* which deals with real numbers and two implementations of complex numbers. Among other things, it includes a *map* function which takes one of each kind of complex into a real,

Transform(*R*: *Real* ×× *C1*: *MakeComplex*(*R*) × *C2*: *MakeComplex*(*R*) →
type) :~ (*map*: (*C1* $ *complex* × *C2* $ *complex* → *R* $ *real*) × ...);

Note that this declaration requires *C1* and *C2* to be based on the same implementation of *Real*. An implementation of this interface would look like

TransformCP: *Transform* (*RealFl*, *Cartesian*, *Polar*) ~
[*map* :~ $\lambda C1$: *Cartesian* $ *complex* × *C2*: *Polar* $ *complex* →
RealFl $ *real*
IN IF ... THEN ... ELSE ..., ...];

Thus in Pebble it is easy to obtain any desired degree of flexibility in defining interfaces and implementations. In most applications, the amount of parameterization shown in these examples is not necessary, and definitions like the simpler ones for *Cartesian* and *Polar* would be used.

We leave it as an exercise for the reader to recast the module facilities of Ada, CLU, Euclid, and Mesa in the forms of Pebble.

3.2. *Abstract Data Types*

An *abstract data type* glues some operations to a type, e.g., a stack with *push*, *pop*, *top*, etc. Clients of the abstraction are not allowed to depend on the value of the type (e.g., whether a stack is represented as a list or an

array), or on the actual implementations of the operations. In Pebble terms, the abstract type is a declaration, and the client takes an implementation as a parameter. Thus

$$\begin{aligned} \mathit{intStackDecl}\text{: type} \sim \quad & (\mathit{stk}\text{: type} \times\times \\ & \mathit{empty}\text{: } \mathit{stk} \times \\ & \mathit{isEmpty}\text{: } (\mathit{stk} \to \text{bool}) \times \\ & \mathit{push}\text{: } (\text{int} \times \mathit{stk} \to \mathit{stk}) \times \\ & \mathit{top}\text{: } (\mathit{stk} \to \text{int}) \times \ldots) \end{aligned}$$

is an abstract data type for a stack of ints. We have used a dependent $\times\times$ type to express the fact that the operations work on values of type *stk* which is also part of the abstraction. We could instead have given a parameterized declaration for the operations

$$\begin{aligned} \mathit{intStackOpsDecl}(\mathit{stk}\text{: type} \to \text{type}) :\sim \\ & (\mathit{empty}\text{: } \mathit{stk} \times \\ & \mathit{isEmpty}\text{: } (\mathit{stk} \to \text{bool}) \times \\ & \mathit{push}\text{: } (\text{int} \times \mathit{stk} \to \mathit{stk}) \times \\ & \mathit{top}\text{: } \mathit{stk} \to \text{int}) \times \ldots) \end{aligned}$$

Matters are somewhat complicated by the fact that the abstraction may itself be parameterized. We would probably prefer a *stack* abstraction, for example, that is not committed to the type of value being stacked. This gives us still more choices about how to arrange things. To illustrate some of the possibilities, we give definitions for the smallest reasonable pieces of a *stack* abstraction, and show various of putting them together.

We begin with a function producing a declaration for the stack operations; it has both the element type *elem* and the stack type *stk* as parameters:

$$\begin{aligned} \mathit{stackOpsDecl}(\mathit{elem}\text{: type} \times \mathit{stk}\text{: type} \to \text{type}) :\sim \\ & (\mathit{empty}\text{: } \mathit{stk} \times \\ & \mathit{isEmpty}\text{: } (\mathit{stk} \to \text{bool}) \times \\ & \mathit{push}\text{: } (\mathit{elem} \times \mathit{stk} \to \mathit{stk}) \times \\ & \mathit{top}\text{: } (\mathit{stk} \to \mathit{elem}) \times \ldots) \end{aligned}$$

With this we can write the previous definition of *intStackOpsDecl* more concisely as

$$\mathit{intStackOpsDecl}(\mathit{stk}\text{: type} \to \text{type}) :\sim \mathit{StackOpsDecl}[\text{int}, \mathit{stk}]$$

The type of a conventional stack abstraction, parameterized by the element type, is a function that produces a declaration for a dependent type:

$$\mathit{StackDecl}(\mathit{elem}\text{: type} \to \text{type}) :\sim \mathit{stk}\text{: type} \times\times \mathit{StackOpsDecl}[\mathit{elem}, \mathit{stk}]$$

and we can write the previous *intStackDecl* as

intStackDecl:type ~ *StackDecl* int

Leaving the element type unbound; we can write an implementation of *StackDecl* using lists to represent stacks,

```
StackFromList(el: type → StackDecl el) :~
                    [stk :~ list el;
                    empty :~ nil;
                    isEmpty(s: stk → bool) :~ s = nil;
                    ...]
WHERE list: type → type ~ ...
```

Here we have given the type of *list* but omitted the implementation, which is likely to be primitive. Then we can apply this to int, getting

IntStackFromList: *IntStackDecl* ~ *StackfromList* int

By analogy with *list*, if we have only one implementation of stacks to deal with we will probably just call it *stack* rather than *StackFromList*. In particular, an ordinary client will probably only use one implementation, and will be written

```
Client(stack: (el: type →» StackDecl el) → ...) :~
                    LET intStack :~ stack int IN
                                    -Client body-
```

This arrangement for the implementation leaves something to be desired in security.

Consider for simplicity the case where we use only integer lists,

```
LET Client(stack: IntStackDecl) :~ -Client body-
  IN ... Client(IntStackFromList)...  comment Main Program;
```

For example,

```
Client(stack: IntStackDecl) :~ (stack;
                    push2(n, s) :~
                        stack $ push(n, stack $ push(n, s)))
  IN LET Stack2 :~ Client(IntStackFromList)
    IN Stack2 $ push2(3, stack2 $ empty))
```

The client body is type-checked without any knowledge of the representation of *stack*, so replacing *stack* $ *push* by *cons* would cause a type error. But the Main Program can construct a list int and pass it off as a *stack2* $ *stack*, so replacing *stack2* $ *empty* by nil would not cause a type error. Any list is an acceptable representation of stacks, but if we had chosen an array with a counter, then passing off an array with a negative

counter would cause disaster. To defend itself against such forgeries, an implementation such as *StackFromList* may need a way to protect the ability to construct a *stk* value. To this end we introduce the primitive

AbstractType: $(T$: type $\times$ *p*: *password* $\rightarrow\!\!\rightarrow$
AT: type $\times\times$ *abs*: $(T \rightarrow AT) \times$ *rep*: $(AT \rightarrow T)) \sim \ldots$;

This function returns a new type AT, together with functions *abs* and *rep* which map back and forth between AT and the parameter type T. Values of type AT can be constructed only by the *abs* function returned by a call of *AbstractType* with the same *Password*.

Other languages with a similar protection mechanism (for example ML) do not use a password, but instead make *AbstractType* non-applicative, so that it returns a different AT each time it is called. This is equivalent to making up a new password automatically each time you recompile. This ensures that no intruder can invoke *AbstractType* on his own and get hold of the *abs* function. We have not used this approach for two reasons. First, a non-applicative *AbstractType* does not fit easily into the formal operational semantics for Pebble. Both the intuitive notion of type-checking described in Section 2 and the formal one in Section 5 depend on the fact that identical expressions in the same environment have the same value, i.e., that all functions are applicative. The use of a password to make an abstract type unique is quite compatible with this approach.

Second, in a system with persistent data, automatic password generation on compilation does not make sense. The implementor might change the implementation of *stack* to make it more efficient without changing the representation. She would not want this to invalidate all existing *stack* values. So the new version would use the old password. Instead we think of converting a value v to an abstract value *abs*(v) as a way of asserting some invariant that involves v. The implementations of operations on *abs*(v) depend on this invariant for their correctness. The implementer is responsible for ensuring that the invariant does in fact hold for any v in an expression *abs*(v); he does this by

- checking that each application of *abs* in his code satisfies a suitable pre-condition;
- preventing any use of *abs* outside his code, so that every application is checked.

A natural way to identify the implementer is by his knowledge of a suitable password. This requires no extensions to the language, and the only assumption it requires about the programming system is that other programmers do not have access to the password in the text of the implementation.

Using *AbstractType* we can write a secure implementation:

```
StackFromList(el: type →» StackDecl el) :~
                    LET (st :~ a $ AT, abs :~ a $ abs, rep :~ a $ rep)
                      WHERE a :~ AbstractType(list el,
                      "PASSWORDXYZ")) IN
                    (stk :~ st;
                    empty :~ abs nil;
                    isEmpty(s: stk → bool) :~ (rep s) = nil;
                        ...)
```

Here we are also showing how to rename the values produced by *AbstractType*; if the names provided by its declaration are satisfactory, we could simply write

```
StackFromList(el: type →» StackDecl el) :~ LET AbstractType(list el,
                      "PASSWORDXYZ") IN
                    (stk :~ AT;
                    empty ~ abs nil;
                    isEmpty(s: stk → bool) :~ (rep s) = nil;
                        ...)
```

The *abs* and *rep* functions are not returned from this *StackFromList*, and because of the password, there is no way to make a type equal to the *AT* which is returned. Hence the program outside the implementation has no way to forge or inspect *AT* values.

Sometimes it is convenient to include the element type in the abstraction:

```
aStackDecl: type ~   elem: type ××
                     stk: type ××
                     StackOpsDecl[elem, stk]
```

This allows polymorphic stack-bashing functions to be written more neatly. An *aStackDecl* value is a binding. For example, redefining *intStack*,

```
intStack: aStackDecl ~ (elem :~ int, StackFromList int)
```

An example of such a polymorphic function is

```
Reverse(S: aStackDecl ×× x: S $stk →» S $ stk) :~ LET S IN
                    LET rev(y: stk × z: stk → stk) :~
                      IF isEmpty y THEN z
                      ELSE rev(pop y, push(top y, z))
                    IN rev(x, empty)
```

so that *Reverse*(*intStack*, *intStack* $ *MakeStack*[1, 2, 3] = *intStack* $ *MakeStack*[3, 2, 1]).

3.3. *Generic Types*

A *generic type* glues a value to an instance of an abstract data type. Thus, for example, we might want a generic type called *atom*, such that each value carries with it a procedure for printing it. A typical *atom* value might be

[string, string $ *Print*, "Hello"]

A simple way to get this effect (using ⟨ ⟩ for string concatenation) is

```
AtomOps(t: type → type) :~   Print: (t → list char);
atomT: type ~                t. type ××
                             AtomOps(t);
atom: type ~                 at: atomT ××
                             val: at $ t;
PrintAtom(a: atom → list char) :~ a $ Print(a $ val);
REC PrintList(l: list atom → list char) :~
    IF null l THEN "[ ]"
    ELSE "[" ⟨ ⟩ PrintAtom(head l) ⟨ ⟩ ","
             ⟨ ⟩ PrintList(tail l) ⟨ ⟩ "]"
```

With this we can write

```
stringAtomT: atomT ~ [string, PrintString];
hello: atom ~ [stringAtomT, "Hello"];
intAtomT: atomT ~ [int, PrintInt];
three: atom ~ [intAtomT, 3]
```

Then *PrintAtom three* = "3", and *PrintList*[*hello*, *three*, *nil*] = "[Hello, [3, [ ]]]."

If int and string are extended types (see Section 2.6) with *Print* procedures, so that xtd ^{1}(*atomT*) succeeds, then we could define *atom* differently:

atom: type ~ *at*: *atomT* ×× *val*: *at* $ *t* ⊗ (xtd^{-1} *atomT*)

Now we can write *PrintAtom*(3), and 3 will be coerced into ((*t* ~ int $ *t*, *Print* ~ int $ *Print*), 3) by the coercion for ⊗ types, because shrinkF(type, *atomT*)(int) evaluates to (*t* ~ int $ *t*, *Print* ~ int $ *Print*).

This is fine for dealing with an individual value which can be turned into a atom, but suppose we want to print a list of ints. It is not attractive to first construct a list of atoms; we would like to do this on the fly. This observation leads to different *Print* functions, using the same definition of

atom. The idea is to package a type t, and a function for turning t's into *atom*s,

```
atomX :~   t: type ×× conv: (t → atom)
PrintAtom(at: atomX ×× v: at $ t → list char) :~
        LET a :~ at $ conv v IN a $ Print(a $ val)
REC Printlist(at: atomX ×× l: list at $ t → list char) :~
        IF null l THEN "[ ]"
        ELSE "[" ⟨⟩ PrintAtom[at, head l] ⟨⟩ ","
                 ⟨⟩ Printlist[at, tail l] ⟨⟩ "]"
intAsAtom: atomX ~ (t :~ int,
                    conv(v: t → atom) :~
                        (t :~ int, Print :~ PrintInt, val :~ v))
```

3.4. *Recursive Types*

Pebble handles recursive functions in the standard operational style, relying on the fact that a λ-expression evaluates to a closure in which evaluates of the body is deferred. The language has types which involve closures, namely the dependent types constructed with →» and ××, and it turns out that the operational semantics can handle recursive type definitions involving these constructors. A simple example is

```
LET REC IntList: type ~ head: int ×× tail: (l: IntList ⊕ v: void)
```

where for simplicity we have confined ourselves to lists of integers rather than introducing a type parameter. Although the evaluation rules for recursion were not designed to handle this kind of expression, they in fact do so quite well. Note that ⊕ has the necessary ×× built in.

3.5. *Assignment*

Although Pebble as we have presented it is entirely applicative, it would be possible to introduce imperative primitives. For example, we could add

```
var: type → type
```

Then var int is the type of a variable whose contents is an int. We also need

```
new: (T: type →» var T) ×
MakeAssign: (T: type →» (var T × T → void)) ×
MakeDereference: (T: type →» (var T → T))
```

From MakeAssign and MakeDereference we can construct := and ↑ procedures for any type.

Of course, these are only declarations, and the implementation will necessarily be by primitives. Furthermore, the semantics given in this paper

would have to be modified to carry around a store which := and ↑ can use to communicate.

In addition, steps would have to be taken to preserve the soundness of the type-checking in the presence of these non-applicative functions. The simplest way to do this is to divide the function types into *pure* or applicative versus *impure* or imperative ones. *MakeAssign* and *MakeDereference* return impure functions, as does any function defined by a λ-expression whose body contains an application of an impure function. Then an impure symbolic value is one that contains an application of an impure function. We can never infer that such a value is equal to any other value, even one with an identical form (at least not without a more powerful reasoning system than the one in the Pebble formal semantics).

4. Values and Syntax

This section gives a formal description of the values and syntax of Pebble. It also defines a relation "has type" (written :::) between *values* and types; in other words, it specifies the set of values corresponding to each type. Note that these sets are not disjoint. Section 5 gives a formal description of the semantics of Pebble, and defines a relation "has type" (written ::) between *expressions* and types.

4.1. *Values*

We start our description of Pebble with a definition of the space of values. These may be partitioned into subsets, such as function values, pairs, and types. Some of these may be further partitioned into more refined subsets, such as cross types and arrow types. Our values are the kind of values which would be handled by a compiler or an interpreter, rather than the ones which would be used in giving a traditional denotational semantics for our language. The main difference is that we represent functions by closures instead of by the partial functions and functionals of denotational semantics. Table II gives a complete breakdown of the set of values.

All these value constructors except "," "!" ":", **closure** and **fix** could be replaced by constants using "!" and ",". Thus, for example, $t \times t$ could become $\times ! (t, t)$.

Each set of values, denoted by a lowercase letter, is composed of the sets written immediately to the right of it, e.g.,

$$e = e_0 \cup f \cup \mathbf{nil} \cup (e, e) \cup b \cup t \cup (f ! e)$$

where by (e, e) we mean the set of all values (v_1, v_2) such that $v_1 \in e$ and $v_2 \in e$. Similarly **nil** means $\{\mathbf{nil}\}$, $(f ! e)$ means $\{(v_1 ! v_2) \mid v_1 \in f, v_2 \in e\}$, and

so on for each value constructing operator. The primitive constants of the value space and constructors such as **closure** are written in **this font** throughout this section. Meta-variables which denote values or sets of values, possibly of a given kind, are single lowercase letters in *this font*, possibly subscripted.

We now examine each kind of value in turn, giving a brief informal explanation. Indented paragraphs describe how a set of values may be partitioned into disjoint subsets.

– e is the set of all values, everything which may be denoted by an expression.

– e_0 consists of the primitive values **true**, **false**, **0**, **1**, ..., all except the functions and types.

– f consists of the values which are functions, as follows.

* The values in f_0 which are primitives such as addition or multiplication of integers. They include the functions $\times$, $\rightarrow$, typeOf on types; the inverse functions $\times^{-1}$, $\rightarrow^{-1}$, $:^{-1}$, $\diamond^{-1}$, $\triangleright^{-1}$, $\otimes^{-1}$; and the functions if, fst, snd, rhs on values. Note that there are no other operations on types,

TABLE II

Values

e	e_0	viz **true, false, 0, 1, 2,** ..., etc.	
	f,	f_0 viz. +, ×, etc.	
		closure(ρ, d, E)	
	nil		
	$[e, e]$		
	b	$n \sim e$	
		nil	
		$[b, b]$	
		fix(f, f)	
	t	t_0 viz, **bool, int,** etc.	
		void	
		$t \times t$	
		$t \diamond f$	
		$t \rightarrow t$	
		$d \triangleright f$	
		$t \otimes f$	
		d	$n{:}\,t$
			void
			$d \times d$
			$d \diamond f$
			$d \otimes f$
	$f!e$		

declarations, or bindings. In particular, there is no equality. This is important if we wish to avoid the need for run-time representations of these things.

* **closure** values, the results of evaluating λ-expressions. A closure is composed of:

1. an *environment* ρ, which associates a type and a value with each name;

2. a *declaration* value, which gives the bound variables of the λ-expresssion;

3. a *body* expression, which is the expression following IN in the λ-expression (expressions are defined in Section 4.2).

– **nil**, the 0-tuple.

– $[e, e]$, the 2-tuples (ordered pairs) of values. The pair forming operation is ",". In general we use brackets for pairs, as in [**1**, [**2**, [**3**, **nil**]]]; formally, brackets are just a syntactic variant of parentheses. Since "," associates to the right, we can also write [**1**, **2**, **3**, **nil**].

– *binding* values, which associate names with values. For example, evaluating LET x: int $\sim 1+2$ IN ... will produce a binding $x \sim 3$ which associates x with **3**. Strictly we should discriminate between "binding expressions" and "binding values," but mostly we will be sloppy and say "binding" for either. Bindings are either elementary or tuples, thus:

* $N \sim e$, which binds a single name N to a value e.

* **nil**. The 0-tuple is also a binding.

* $[b, b]$, which is a pair of bindings, is also a binding. The binding $[b_1, b_2]$ binds the variables of b_1 and those of b_2. This is a special case of $[e, e]$ above, since b is a subset of e.

* **fix** values, which result from the evaluation of recursive bindings. A **fix** value contains the function which represents one step of the recursive definition (roughly, the functional whose fixed point is being computed). Details are given in Section 5.2.5.

– *type* values, consisting of:

* t_0, some built-in types such as booleans (**bool**) and integers (**int**). They include the type **type** which is the type of all type expressions.

* **void**, the type of **nil**.

* $t \times t$, which is the type of pairs. If expression E_1 has type t_1 and expression E_2 has type t_2, then the pair $[E_1, E_2]$ has type $t_1 \times t_2$.

* $t \diamond f$, a dependent version of $t \times t$. This is explained in Section 2.5.

* $t \rightarrow t$, which is the type of functions.

* $d \triangleright f$, a dependent version of $t \to t$. This is explained in Section 2.4.

* $t \otimes f$, the inferred product type. This is explained in Section 2.6.

* d, declarations. These are the type of bindings; for example, the type of x: int $\sim 1+2$ is x: **int**. They give types for the three kinds of bindings above.

– N: t, a basic declaration, which associates name N with type t, e.g., x: **int**.

– **void**, the type of the nil binding.

– $d \times d$, the type of a pair of bindings (a special case of $t \times t$).

– $d \diamond f$, a dependent version of $d \times d$.

– $d \otimes f$, the inferred product type.

– $f ! e$ is the application of the primitive function or symbolic value f to the value e. Such applications are values which may be simplified.

To formulate a soundness theorem we may define a relation ::: between values and types, analogous to the :: ("has type") relation between expressions and types defined in Section 5. Unlike the latter, it is independent of any environment. We could define it by operational semantic rules, but it is shorter to give the following informal inductive definition. In one or two places we need the $\Rightarrow$ ("has value") relation between expressions and values defined in Section 5. We first define a subsidiary function typeOf from declaration values to type values; for example, typeOf(x: **int**) = **int** and typeOf(x: **int** $\times$ p: **bool**) = **int** $\times$ **bool**,

$$\text{typeOf}(\textbf{void}) = \textbf{void}$$

$$\text{typeOf}(N : t) = t$$

$$\text{typeOf}(d_1 \times d_2) = \text{typeOf}(d_1) \times \text{typeOf}(d_2).$$

(The cases for $\diamond$ and $\otimes$ are given in Table VI.)

We also need a notion of applying a funcion value f to an argument value e_0 to obtain a result value e; this is written $f ! e_0 \rightsquigarrow e$ and is defined precisely in Section 5. For example, $+ ! (3, 4) \rightsquigarrow 7$.

Now for the definition of ::: which relates values and types,

– **true** ::: **bool**, **false** ::: **bool**, **0** ::: **int**, **1** ::: **int**, and so on.

not ::: **bool** $\to$ **bool**, and so on for other operators.

$\times$ ::: **type** $\times$ **type** $\to$ **type**,

$\to$ ::: **type** $\times$ **type** $\to$ **type**.

closure$(\rho, d, E) ::: t_1 \to t_2$ if $t_1 =$ typeOf d and for all bindings b such that $b ::: d$ we have $\rho[d \sim b] \vdash E :: t_2$.

– **nil** ::: **void**.

$e_1, e_2 ::: t_1 \times t_2$ if $e_1 ::: t_1$ and $e_2 ::: t_2$.

– $N \sim e ::: N : t$ if $e ::: t$.

fix$(f, f) ::: d$ if $f ::: d \to d$.

– **bool** ::: **type**, **int** ::: **type**, **type** ::: **type**, **void** ::: **type**.

$t_1 \times t_2 ::: \mathbf{type}$ if $t_1 ::: \mathbf{type}$ and $t_2 ::: \mathbf{type}$.

– $N : t ::: \mathbf{type}$ if $t ::: \mathbf{type}$.

$d \diamond f ::: \mathbf{type}$ if $f ::: d \to \mathbf{type}$.

$d \rhd f ::: \mathbf{type}$ if $f ::: d \to \mathbf{type}$.

$t_1 \to t_2 ::: \mathbf{type}$ if $t_1 ::: \mathbf{type}$ and $t_2 ::: \mathbf{type}$.

$f!\, e ::: t_2$ if $e ::: t_1$ and $f ::: t_1 \to t_2$.

– $e_1, e_2 ::: t_1 \diamond f$ if there is some t_2 such that $f!\, e_1 \rightsquigarrow t_2$ and $(e_1, e_2) ::: t_1 \times t_2$.

$e ::: t \otimes f$ if $e ::: t$.

$f_1 ::: d \rhd f$ if for all e_0 such that $e_0 ::: d$, there is some t_2 such that $f!\, e_0 \rightsquigarrow t_2$ and $f_1!\, e_0 \rightsquigarrow e$ and $e ::: t_2$.

Now if $E \Rightarrow e$, we would like to have $e ::: t$ if $E :: t$ (soundness); we hope

TABLE III

Syntax

Type		Introduction	Elimination
T bool			IF E THEN E_1 ELSE E_2
$T_1 \times T_2$		E_1, E_2	fst E snd E
			E_1 AS E_2
$T \to T_0$	F	$\lambda\, T$ IN E	$F\, E$
$D \to T_0$		Primitives	
D $N: T$	B	$D \sim E$	LET B IN E
$D_1 \times D_2$		REC $D \sim E$, ..., $D \sim E$	IMPORT B IN E
$D_1 \times\times D_2$		B_1, B_2	
$F \otimes D$		$B_1; B_2$	
		$N: \sim E$	
		$N(T): \sim E$	
type		all types in the left column	typeOf D
		N	

Note. Either round or square brackets may be used for grouping. All the operators associate to the right. Precedence is: lowest IN, then "," ";", then $\sim$, then $\to$ $\to\!\!>$, then $\times$ $\times\times$, then: highest application.

this is provable. But our type-checking rules, which use symbolic evaluation, cannot always achieve $E :: t$ if $e ::: t$ (completeness). A closure may have a certain type for all bindings, but symbolic evaluation may fail to show this. Consider for example

$(\lambda x$: int →» (IF $x \leqslant x+1$ THEN int ELSE bool) IN $x) ::$ **int → int**

This is not derivable from our type-checking rules because symbolic evaluation cannot show that $x \leqslant x+1$ for an arbitrary integer x. But the latter is true, so if f is the value of the lambda expression we do get $f ::$ **int → int** by the definition above for closures. This limitation does not seem to present a major practical obstacle, but the matter would repay further study.

4.2. *Syntax*

We can give the syntax of Pebble in traditional BNF form, but there will be only three syntax classes: name (N), number (I), and expression (E),

N ::= letter(letter | digit)*
I ::= digit digit*
E ::= E →» E | N : E | E ×× E | [E, E] |
λE IN E | N(E) :~ E | FIX E | REC E ~ E | N :~ E |
E; E | E $ N | E.N | N | IMPORT E IN E
| IF E THEN E ELSE E | E AS E | E E | LET E IN E | (E) | [E].

It is more helpful to divide the expressions up according to the type of value they produce. We distinguish subsets of the set E of all expressions thus: T for types, D for declarations, B for bindings, and F for functions. These cannot be distinguished syntactically since an operator/operand expression of the form E E could denote any of these, as could a name used as a variable. However, it makes more sense if we write, for example, LET B IN E instead of LET E IN E, showing that LET requires an expression whose value is a binding.

TABLE IV

Summary of Abbreviations

Non-terminal	Must evaluate to	Example
N	Name	*i*
E	Expression	*gcd*(*i*, 3) + 1
T	Type	int
D	Declaration	*i*: int
B	Binding	*i*: int ~ 3
F	Function	λ *i*: int → bool IN *i* > 3

All the non-terminals except N are syntactically equivalent to E.

TABLE V

Sugar

Write	For	Example
$N: \sim E$	$N: t \sim E$ (provided $E: > t$)	$i: \sim 3$
$(N_1, ..., N_n): \sim E$	$(N_1 \sim \text{fst } E, (N_2, ..., N_n): \sim \text{snd } E))$	$(i, j): \sim (3, 4)$
$N(T): \sim E$	$N: \sim \lambda\ T$ IN E	$sq(i: \text{int} \to \text{int}): \sim i * i$
$\lambda\ D$ IN E	$\lambda\ D \to t$ IN E (provided LET newc $_{*d}$ IN $E: > t$ and newc doesn't appear in t)	$\lambda\ i$: int IN $i * i$
$B_1; B_2$	B_1, LET B_1 IN B_2	$i: \sim 3; j: \sim i + 2$
B S N	IMPORT B IN N	$(i: \sim 3, j: \sim 4)$ S $i \equiv 3$
E WHERE B	LET B IN E	$i + 1$ WHERE $i: \sim 3$
$D \to\to T$	$D \rhd (\lambda\ B': D \to \text{type}$ IN LET B' IN $T)$	t: type $\to\to (t \to t)$
$D \times\times T$	$D \diamond (\lambda\ B': D \to \text{type}$ IN LET B' IN $T)$	b: bool $\times\times$ IF b THEN int ELSE real
REC $D_1 \sim E_1, ..., D_n \sim E_n$	FIX$((\lambda\ B': D_1 \times \cdots \times D_n$ IN LET B' IN $D_1 \sim E_1), ...,$ $(\lambda\ B': D_1 \times \cdots \times D_n$ IN LET B' IN $D_n \sim E_n))$	
REC $N_1(T_1): \sim E_1, ...$	REC $N_1: T_1 \sim \lambda\ T_1$ IN $E_1, ...$	REC $fact(i: \text{int} \to \text{int}): \sim$ IF $i = 0$ THEN 1 ELSE...
$E.N$	(snd $\text{xt}^{-1}\ t) \uparrow$ S $N(E)$ (provided $E: > t$)	$s.pop$ where $s.Stack$
$E_1.N(E_2)$	(snd $xt^{-1}\ t) \uparrow$ S $N(E_1, E_2)$(provided $E: > t$)	$s.push(3)$ where s: $Stack$

It is also helpful to organise the syntax according to types and to the introduction and elimination rules for expressions of each type. This is a common format in recent work on logic. For example, a value of type $T_1 \times T_2$ is introduced by an expression of the form E_1, E_2; it is eliminated by expressions of the form fst E or snd E.

The syntax presented in this way is shown in Table III; a list of the notations used is given in Table IV. Table V shows some abbreviations which make Pebble more readable, for example eliminating the λ notation for function definitions in traditional style.

5. Operational Semantics

We have a precise operational semantics for Pebble, in the form of the set of inference rules in Table VI. This section gives the notation for the inference rules, explains why they yield at most one value for an expression, and discusses the way in which values can be converted into expressions and fed back through the inference system. Then we explain in detail how each rule works.

TABLE VIa

Inference Rules for $\Rightarrow$: Introduction

Rule	Type		Introduction
×I	void $T_1 \times T_2$	(1)	$E_1 :: t_1, E_2 :: t_2$
		(0)	$[E_1, E_2] :> t_1 \times t_2 \Rightarrow [e_1, e_2]$
→I	$T_0 \to T$ $D \to T$ d = parameter decl t_0 = parameter type t = result type t_1 = type of λ-exp	(1) (2) (3) (4)	$\{T_1 \Rightarrow t_1', t_1' \approx d \to t, \text{typeOf d} \Rightarrow t_0, t_0 \to t = t_1$ or $T_1 \Rightarrow t_1, t_1 \approx d \rhd f, \text{f!newc}(n+1) \rightsquigarrow t\}$, $\rho(\text{depth}) = n, \rho[\text{depth} = n+1] = \rho'$ $\rho' \vdash \text{LET newc}(n+1)_{\neq d} \text{ IN E} :: t$
		(0)	$(\lambda\, T_1 \text{ IN } E) :> t_1 \Rightarrow \text{closure}(\rho', d, E)$
:I	$N : T$ $D_1 \times D_2$ $D_1 \times\times D_2$	(1)	$F :: t, \text{fixtype}(t) \rightsquigarrow t', t' \approx d \to d$
		(0)	$\text{FIX } F :: d \Rightarrow \text{fix}(f, f)$
typeI	type	(1)	$T :: \text{type}$
		(0)	$N : T :> \text{type} \Rightarrow N : t$
NI	Names	(1)	$\rho(N) \approx t \sim e_0, e_0 \rightsquigarrow e$
		(0)	$N :> t \Rightarrow e$

TABLE VIb

Inference Rules for ⇒: Elimination

Rule	Type		Elimination
boolE	bool int	(1) (2)	E_0 :: bool, E_1 :: t, E_2 :: t, {$E_0 \Rightarrow$ true, $E_1 \Rightarrow e$ **or** $E_0 \Rightarrow$ false, $E_2 \Rightarrow e$ else if!(e_0, e_1, e_2) = e}
		(0)	IF E_0 THEN E_1 ELSE E_2 :> t ⇒ e
× E	void $T_1 \times T_2$	(a0)	fst :: $(t \times t_1) \to t$ (b0) snd :: $(t_1 \times t) \to t$
		(c1) (c2)	E_1 :: $d \diamond f$, $f_{\neq d \to type}(d \sim E_2) \Rightarrow t$, snd $E_1 \Rightarrow e_{12}$ fst $E_1 = E_2 \Rightarrow e'$, {e' = true, $e_{12} = e$ **or** e' ≃ e_1'! e_2', iftrue!(e', e_{12}) = e}
		(c0)	E_1 AS E_2 :> t ⇒ e
→E	$T_0 \to T$ $D \twoheadrightarrow T$ t_0 = parameter type e_0 = argument value t = result type	(1) (2) (3)	{F :: $t_0 \to t$, coerce(E_0, t_0) :: $t_0 \Rightarrow e_0$, **or** F :: $d \rhd f_1$, $f_{1 \neq d \to type}(d \sim E_0) \Rightarrow t$, rhs($d \sim E_0$) ⇒ e_0}, {f! $e_0 \rightsquigarrow e$ else f! $e_0 = e$}
		(0)	$F E_0$:> t ⇒ e
:E	$N : T$ $D_1 \times D_2$ $D_1 \times\times D_2$	(1) (2) (3) (4)	B :: void, E :> $t \Rightarrow e$ or B :: $(N : t_0)$, rhs B ⇒ e_0, $\rho[N = t_0 \sim e_0] \vdash$ E :> $t \Rightarrow e$ or B :: $d_1 \times d_2$, snd B ⇒ b_2, LET fst B IN LET $b_{2 \neq d_2}$ IN E :> $t \Rightarrow e$ or B :: $d_1 \diamond f$, $f_{\neq d_1 \to type}$(fst B) ⇒ d_2, LET $b_{\neq d_1 \times d_2}$ IN E :> $t \Rightarrow e$
		(0)	LET B IN E :> t ⇒ e
		(a1)	B :: $d \Rightarrow b$, [ ] ⊢ LET $b_{\neq d}$ IN E :: $t \Rightarrow e$ (b1) typeOf!$d \rightsquigarrow t$
		(a0)	IMPORT B IN E :> t ⇒ e (b0) rhs :: d → t
typeE	type	(1) (2)	E :: t, E ⇒ e' else E :: t', coerce$F$(t', t) $\rightsquigarrow f$, f! e $\rightsquigarrow e'$
		(0)	coerce(E, t) :: t ⇒ e'

TABLE VIc

Inference Rules for $\Rightarrow$: others

Rule		
::	(1)	$E :> t$
	(2)	or $\{t \approx t_1 \diamond f, E :: t_1 \times t_2$ or $t \approx t_1 \times t_2, E :: t_1 \diamond f\}, f_{\#t_1 \to type}$ fst $E) \Rightarrow t_2$
	(3)	or $E :: t \otimes f$ or $t \approx t' \otimes f, E :: t'$
	(4)	or $E :: t', \{t' \approx xt!(d, (N \sim t, b))$ or $t \approx xt!(d, (N \sim t', b))\}$
	(0)	$E :: t$
#	(0)	$e_{\#t} :> t \Rightarrow e$
	(s0)	$e :> type \Rightarrow e$

TABLE VId

Inference Rules for $\rightsquigarrow$ and $\rightsquigarrow\!\!\!\!\rightarrow$

$\rightsquigarrow$ For each ⟨arg, result⟩ pair in each primitive f_0

(0) $f_0!\, e_0 \rightsquigarrow e$ (a0) $fst![e, e_1] \rightsquigarrow e$ (b0) $snd![e_1, e] \rightsquigarrow e$

(c1) $e' = \text{nil}, \text{nil} = e$
(c2) **or** $e' \approx N \sim e$
(c3) **or** $e' \approx [e'_1, e'_2]$, rhs!$e'_1 \rightsquigarrow e_1$, rhs!$e'_2 \rightsquigarrow e_2$, $[e_1, e_2] = e$

(c0) rhs!$e' \rightsquigarrow e$

(d1) typeOf!$d \rightsquigarrow t$, $\rho' \vdash$ LET $d \sim e_{0\,\neq t}$ IN $E \Rightarrow e'$

(d0) closure(ρ', d, E)!$\, e_0 \rightsquigarrow e'$

(e1) $d \approx \text{void}, \text{void} = t$
(e2) **or** $d \approx N{:}t$
(e3) **or** $d \approx d_1 \times d_2$, typeOf!$d_1 \rightsquigarrow t_1$, typeOf!$d_2 \rightsquigarrow t_2$, $t_1 \times t_2 = t$
(e4) **or** $d \approx d_1 \diamond f_2$, typeOf!$d_1 \rightsquigarrow t_1$, $\rho_0 \vdash (\lambda B'{:}\, d_1$ IN typeOf $f_{2\,\neq d_1 \to \text{type}}\, B') \Rightarrow f'_2$, shrinkF!$(d, d) \rightsquigarrow f$, $t_1 \diamond f'_2 = t$
(e5) **or** $d \approx d_1 \otimes f_2$, typeOf!$d_1 \rightsquigarrow t_1$, $d_1 \approx d_{11} \diamond f_{12}$, $\rho_0 \vdash (\lambda N'{:}$ type IN rhs $f_{2\,\neq \text{type} \to d_{11}}\, N') \Rightarrow f'_2$, $t_1 \otimes f'_2 = t'$

(e0) typeOf!$d \rightsquigarrow t$

(f1) $d \approx \text{void}, \text{nil} = b$
(f2) **or** $d \approx N{:}t$, $(N \sim e) = b$
(f3) **or** $\{d = d_1 \times d_2$ **or** $d \approx d_1 \diamond f_2, f_2!\, e_1 \rightsquigarrow d_2\}$, $\{e \approx [e_1, e_2]$ **else** fst!$e = e_1$, snd!$e = e_2\}$, $d_1 \sim e_1 \rightsquigarrow b_1$, $d_2 \sim e_2 \rightsquigarrow b_2$, $[b_1, b_2] = b$

(f4) or $d \approx d_1 \otimes f_2$, $d_1 \sim e \rightsquigarrow b$

(f0) $d \sim e \rightsquigarrow b$

(g1) $t \approx d \to d'$

(g2) or $t \approx t_1 \times t_2$, fixtype(t_1) $\rightsquigarrow d \to d'_1$, fixtype($t_2$) $\rightsquigarrow d \to d'_2$, $d'_1 \times d'_2 = d'$

(g0) fixtype(t) $\rightsquigarrow d \to d'$

(b1) $t \approx t_1 \times t_2$, $\rho_0 \vdash \lambda N' : t'$ IN (coerce(fst N', t_1), coerce(snd N', t_2)) $\Rightarrow f$

(b2) or $t \approx t_1 \diamond f_2$, $\rho_0 \vdash \lambda N' : t'$ IN LET $N'_1 :\sim$ coerce(fst N', t_1) IN (N'_1, coerce(snd N', $f_{2 \neq t_1 \to \text{type}}\ N'_1$)) $\Rightarrow f$

(b3) or $t \approx t_1 \to t_2$, $t' \approx t'_1 \to t'_2$, $\rho_0 \vdash \lambda N' : t'$ IN $\lambda N'' : t_1$ IN coerce(N'(coerce(N'', t'_1)), t_2) $\Rightarrow f$

(b4) or $t \approx d_1 \rhd f_2$, $t' \approx d'_1 \rhd f'_2$, $\rho_0 \vdash \lambda N' : t'$ IN $\lambda N'' :$ typeOf d_1 IN coerce(N'(coerce(N'', typeOf d'_1)), $f_{\neq d_1 \to \text{type}}(d_1 \sim N'')$) $\Rightarrow f$

(h5) or $t \approx N : t_0$, $\rho_0 \vdash \lambda N' : t'$ IN $t \sim$ rhs $N' \Rightarrow f$

(h6) or $t \approx t_1 \otimes f_2$, fstt $t_1 \Rightarrow t_{11}$, $f_2!\ t' \rightsquigarrow e_{11}$, $\rho_0 \vdash \lambda N' : t'$ IN coerce($(e_{11 \neq t_{11}}, N')$, t) $\Rightarrow f$

(h7) or shrinkF(t', t) $\Rightarrow f$

(h8) or t = type, $\rho_0 \vdash \lambda N' : t' \to$ type IN LET T' : type $\sim$ rhs fst N' IN (fst N', snd N') $\Rightarrow f$

(h0) coerceF!(t', t) $\rightsquigarrow$ f

(i1) $d \approx$ void, $\rho_0 \vdash \lambda B' :$ d' IN nil $\Rightarrow f$

(i2) or $d \approx N : t$, $\rho_0 \vdash \lambda B' :$ d' IN d $\sim$ (B'SN) $\Rightarrow f$

(i3) or $d \approx d_1 \times d_2$, $\rho_0 \vdash \lambda B' :$ d' IN (shrinkF(d', d_1) B', shrinkF(d', d_2) B') $\Rightarrow f$

(i4) or $d \approx d_1 \diamond f_2$, $\rho_0 \vdash \lambda B' :$ d' IN LET B'' :$\sim$ shrinkF(d', d_1) B' IN (B'', shrink(d', $f_{2 \neq d_1 \to \text{type}}$B'') B') $\Rightarrow f$

(i0) shrinkF!(d', d) $\rightsquigarrow$ f

$\rightsquigarrow$ (1) $e \approx w!e_1$, $e_1 \rightsquigarrow e_2$, {w!$e_2 \rightsquigarrow e'$ or w!$e_2 \rightsquigarrow e'$}

(2) or $e \approx w!$fix(f', f), {w!$f' \rightsquigarrow f''$, fix(f'', f) = e'

(3) or $f' \approx$ closure(x), $f'!$fix(f', f) $\rightsquigarrow e''$, w!$e'' \rightsquigarrow e'$}

---

(0) $e \rightsquigarrow e'$

5.1. *Inference Rule Semantics*

The basic idea, which we derive from Plotkin, is to specify an operational semantics by means of a set of inference rules. The operations of evaluation are the steps in a proof that uses the rules. The advantage of this approach is that the control mechanism of the evaluator does not need to be written down, since it is implicit in the well-known algorithm for deriving a proof. Indeed, our rules can be trivially translated into Prolog, and then can be run to give a working evaluator. This has been done by Glen Stone, a student at Manchester University, for a slightly different version of the rules.

In general, of course, this will lead to a non-deterministic and inefficient evaluator; the particular rules we use, however, allow an efficient deterministic evaluator to be easily derived.

5.1.1. *Notation.* Each rule has a set of premises $\text{assertion}_1, \ldots, \text{assertion}_n$ and a conclusion assertion_0, written thus:

$$\frac{\text{assertion}_1, \ldots, \text{assertion}_n}{\text{assertion}_0}.$$

As usual, the meaning is that if each of the premises is established, then the conclusion is also established. We write

$$\frac{\text{assertion}_{11}, \ldots, \text{assertion}_{1n_1} \textbf{ or } \text{assertion}_{21}, \ldots, \text{assertion}_{2n_2}}{\text{assertion}_0}$$

as an abbreviation for the two rules

$$\frac{\text{assertion}_{11}, \ldots, \text{assertion}_{1n_1}}{\text{assertion}_0} \qquad \frac{\text{assertion}_{21}, \ldots, \text{assertion}_{2n_2}}{\text{assertion}_0}.$$

Note that **or** has lower precedence than ",". Sometimes **or** is more deeply nested, in which case the meaning is to convert the premises to disjunctive normal form, and then apply this expansion.

An assertion is

environment ⊢ simple assertion.

An *environment* is a function mapping a name to a type and a value. The environment for the conclusion is always denoted by ρ, and is not written explicitly. If the environment for a premise is also ρ (as it nearly always is), it is also omitted.

A *simple assertion* is one of the following.

(1a) $E :: t$ asserts that E has type t in the given environment.

(1b) $E :> t$ asserts that E has principal type t in the given environment.

(2) $E \Rightarrow e$ asserts that E has value e in the given environment.

(3) $e \approx format$ asserts that e is of the form given by format, i.e., that after each variable in format is replaced by some sequence of symbols, the resulting sequence of symbols is identical to e. Every occurrence of a variable in a rule must be instantiated in the same way. For example, $e \approx t_1 \to t_2$; here $t_1 \to t_2$ is a format, with variables t_1 and t_2. If e is int → bool, this assertion succeeds with $t_1 = \text{int}$ and $t_2 = \text{bool}$.

There are four forms of simple assertion which are convenient abbreviations:

(4a) $E :: t \Rightarrow e$ combines (1a) and (2a)

(4b) $E :> t \Rightarrow e$ combines (1b) and (2)

(5) $E :: format$ combines (1a) and (3); it is short for $E :: t$, $t \approx format$.

(6) $e_1 = e_2$ asserts that e_1 is equal to e_2; this is a special case of (3).

Finally, there are two forms of simple assertion which correspond to introducing auxiliary functions into the evaluator:

(7) $e_1 \rightsquigarrow e_2$ asserts that e_1 *simplifies* to e_2, using the simplification rules which tell how to evaluate primitives. See Section 5.2.2.

(8) $e_1 \rightsquigarrow\!\!\!\!\rightarrow e_2$ asserts that e_1 *unrolls* to e_2, using the rule for unrolling fix. See Section 5.2.5.

By convention we write a lowercase e for the value of the expression E, and likewise for any other capital letter that stands for an expression. If a lowercase letter x appears in an assertion, X appears on the left hand side in the conclusion, and no premise has the form $\ldots = x$ or $\ldots \Rightarrow x$, then the premise $X \Rightarrow x$ is implied.

A reminder of our typographic conventions: We use capital letters for meta-variables denoting expressions, and lowercase letters for meta-variables denoting values; both may be subscripted. Thus expressions appear on the left of ::, :>, and ⇒ in assertions, and values everywhere else.

The value constructors that are not symbols are closure and fix.

An italicized meta-variable indicates where that variable will be bound by a deterministic evaluator, as explained in the next section.

5.1.2. *Determinism.* In order to find the principal type of an expression E, we try to prove $E :> t$, where t is a new meta-variable. If a proof is

possible, it yields a value for t as well. Similarly, we can use the inference rules to find the value of E by trying to prove $E \Rightarrow e$. We would like to be sure that an expression has only one value (i.e., that $E \Rightarrow e_1$ and $E \Rightarrow e_2$ implies $e_1 = e_2$). This is guaranteed by the fact that the inference rules for evaluation are *deterministic*: at most one rule can be applied to evaluate any expression, because there is only one conclusion for each syntactic form. When there are multiple rules abbreviated with or, the first premise of each rule excludes all the others. In a few places we write

$a_{11}, ..., a_{1n_1}$ **or** $a_{21}, ..., a_{2n_2}$ **or** ... **or** $a_{k1}, ..., a_{kn_k}$ **else** $a_2, ..., a_n$

as an abbreviation for

$a_{11}, ..., a_{1n_1}$ **or** $a_{21}, ..., a_{2n_2}$ **or** ... **or** $a_{k1}, ... a_{kn_k}$
or not a_{11}, **not** $a_{21}, ...,$ **not** $a_{k1}, a_2, ..., a_n$.

The fact that the rules are deterministic is important for another reason: they define a reasonably efficient deterministic program for evaluating expressions.

Not only has an expression just one value, but it also has just one principal type (defined by the :> relation). It is not true, however, that an expression has only one type. In particular, the auxiliary rule :: may allow types to be inferred for an expression in addition to the principal type. We say more about what this means for deterministic evaluation in Section 5.2.6.

In each rule one occurrence of each meta-variable is italicized. This is the one which the deterministic evaluator will use to bind the meta-variable. For example, in $\times$I1, t_1 and t_2 are bound to the types of E_1 and E_2, respectively; they are used in $\times$I0 to compute $t_1 \times t_2$, the type of $[E_1, E_2]$. The italic occurrence of e may be omitted if it is $E \Rightarrow e$, as explained earlier. Thus the e_1 and e_2 in $\times$I0 are bound by omitted premises $E_1 \Rightarrow e_1$ and $E_2 \Rightarrow e_2$. The italics are not part of the inference rules, but are just a comment which is relevant for deterministic evaluation, and may be a help to the reader as well.

It may also be helpful to know theat the premises are written in the order that a deterministic evaluator would use. In particular, each meta-variable is bound before it is used. In this ordering, the expression in the conclusion should be read first, then the premises, and then the rest of the conclusion.

5.1.3. *Feedback.* An important device for keeping the inference rules compact is that a value with a known type can be converted into an expression, which can then be embedded in a more complex expression whose type and value can be inferred using the entire set of rules. This *feedback* from the value space to the expression space is enabled by the syntax $e \# t$.

This is an expression which has value e and type t. This form of

expression is *not* part of the language, but is purely internal to the inference rules. Usually the type is not interesting, although it must be there for the feedback to be possible, so we write such an expression with the type in a small font, $e_{\#t}$, to make it easier for the reader to concentrate on the values. If t is omitted, it is assumed to be type. In addition, we often drop the $\#t$ entirely in the text of the paper, where no confusion is possible.

5.1.4. *Initial environment.* The expression which constitutes the entire program is evaluated in the initial environment ρ_0 given in Table VII. This provides meaning for standard constants such as true and type, and for standard operators such as × and typeOf.

5.2. *The Rules*

The inference rules in Table VI are organized like the syntax in Table III, according to the expression forms for introducing and eliminating values of a particular type. A particular rule is named by the constructor for the type, followed by I for introduction or E for elimination; thus →I is the rule for λ-expressions, which introduce function values with types of the form $t_0 \to t$. Each line is numbered at the left, so that, for example, the conclusion of the rule for λ-expressions can be named by →I0. If there is more than one rule in a part of the table labelled by the same name, the less important ones are distinguished by letters *a*, *b*, ...; thus × Ec is the rule for AS. Auxiliary rules, with conclusions which are not part of the syntax, appear overleaf. Most of these define the $\rightsquigarrow$ function for simplifying values.

5.2.1. *Booleans, pairs, and names.* The inference rules for booleans are extremely simple,

boolE (1) $E :: \text{bool}, E_1 :: t, E_2 :: t,$

(2) $\{E_0 \Rightarrow \text{true}, E_1 \Rightarrow e \text{ or } E_0 \Rightarrow \text{false}, E_2 \Rightarrow e \text{ else if!}(e_0, e_1, e_2)\}$

(0) $\text{IF } E_0 \text{ THEN } E_1 \text{ ELSE } E_2 :> t \Rightarrow e$

The boolE rule says that the expression

IF E_0 THEN E_1 ELSE E_2

type-checks and has type t if E_0 has type bool, and E_1 and E_2 both have type t for some t. The value of the IF is the value of E_1 if the value of E_0 is true, the value of E_2 if the value of E_0 is false. If the value of E_0 is not known, the IF evaluates to a symbolic value (unless of course it fails to terminate). Thus

(A) IF true THEN 3 ELSE 5

has type int and value 3. The types and values for the constants true, 3, and 5 come from ρ_0.

TABLE VII

Initial Environment ρ_0

Name	Type	Value
true	bool	true
false	bool	false
0, 1, ...	int	0, 1, ...
+, −, *,	int × int → int	+, −, *,
void	type	void
nil	void	nil
×, →	type × type → type	×, →
typeOf	type → type	typeOf
~	d : type ×× typeOf d →» d	
▷, ◇	t : type ×× (t → type)	▷, ◇
⊗	t : type ×× (type → fstt t) → type	⊗
fstt	type → type	λt : type IN fst $\times^{-1} t$
sndt	type → type	λt : type IN snd $\times^{-1} t$
$\times^{-1}$, $\rightarrow^{-1}$	type → type × type	$\times^{-1}$, $\rightarrow^{-1}$
$\triangleright^{-1}$, $\diamond^{-1}$	type → t : type ×× (t → type)	$\triangleright^{-1}$, $\diamond^{-1}$
$\otimes^{-1}$	type → t : type ×× (type → fstt t)	$\otimes^{-1}$
shrinkF	type × type → type	shrinkF
coerceF	type × type → type	coerceF
xt	d : type ×× d → type	xt (d must be declaration)
xt^{-1}	t : type →» typeOf(d : type ×× b : d)	xt^{-1}
xtd^{-1}	d : type →» (t : type →» d)	λd : type IN λt : type IN LET (d', b) : ~ $xt^{-1} t$ IN shrinkF(d', d) b

Note. The following primitives are not in the initial environment, but are generated by the inference rules:

if!(e_0, e_1, e_2) ⇝ e_1 if e_0 = true, e_2 if e_0 = false

iftrue!(e_0, e_1) ⇝ e_1 if e_0 = true, undefined if e_0 = false

fst,snd,rhs with meanings given by the ⇝ rules in 5.

ρ_0 maps each name to type ~ value. It also maps the symbol depth to 0.

We can display this argument more formally as an upside-down proof, in which each step is explicitly justified by some combination of already justified steps, denoted by numbers, and inference rules, denoted by their names (together with some meta-rules which are not mentioned explicitly, such as substitution of equals for equals).

(A1) IF true THEN 3 ELSE 5 :: int ⇒ 3 2,3,4,boolE

(A2) true :: bool ⇒ true NI

(A3) 3 :: int ⇒ 3 NI

(A4) 5 :: int. NI

In this display we show the conclusion at the top, and successively less

difficult propositions below it. Viewing the inference rules as a (deterministic) evaluation mechanism, each line shows the evaluation of an expression from the values of its subexpressions, which are calculated on later lines. Control flows down the table as the interpreter is called recursively to evaluate sub-expressions, and then back up as the recursive calls return results that are used to compute the values of larger expressions.

The rules for pairs are equally simple.

$$\times \mathrm{I} \qquad \frac{(1)\ \mathrm{E}_1 :: t_1, \mathrm{E}_2 :: t_2}{(0)\ [E_1, E_2] :> t_1 \times t_2 \Rightarrow [e_1, e_2]}$$

$$\times \mathrm{E} \qquad \overline{(\mathrm{a0})\ \mathrm{fst} :: (t \times t_1) \to \mathrm{t}} \quad \overline{(\mathrm{b0})\ \mathrm{snd} :: (t_1 \times t) \to \mathrm{t}}.$$

$\times$I says that the type of $[E_1, E_2]$ is $t_1 \times t_2$ if t_i is the type of E_i, and its value is $[e_1, e_2]$. $\times$E gives the (highly polymorphic) types of the primitives fst and snd that decompose pairs.

The rule for names is also straightforward, except for the $\rightsquigarrow$ clause which is treated in Section 5.2.5 since it is needed only for recursion.

$$\mathrm{NI} \qquad \frac{(1)\ \rho(\mathrm{N}) \approx t \sim e}{(0)\ N :> \mathrm{t} \Rightarrow \mathrm{e}}$$

We can use NI to show

$[i = \mathrm{int} \sim 3] \vdash$ IF true THEN i ELSE 0 :: int $\Rightarrow$ 3

following the proof of (A) above, but replacing (A3) with

(A3′) $[i = \mathrm{int} \sim 3] \vdash i :: \mathrm{int} \Rightarrow 3.$ NI

5.2.2. *Functions.* The pivotal inference rules are $\to$I (for defining a function by a λ-expression) and $\to$E (for applying a function). The $\to$I rule is concerned almost entirely with type-checking. If the type-checks succeed, it returns a closure which contains the current environment ρ, the declaration d for the parameters, and the unevaluated expression E which is the body of the λ-expression. A later application of this closure to an argument E_0 is evaluated (using $\to$E) by evaluating the expression

$$\mathrm{LET}\ d \sim E_0\ \mathrm{IN}\ E \qquad (1)$$

in the environment ρ which was saved in the closure.

We begin with the basic rule for λ, omitting line 2, which deals with dependent function types:

(1) $\mathrm{T}_1 \Rightarrow t_1'$, $\mathrm{t}_1' \approx d \to t$, typeOf d $\Rightarrow t_0$, $\mathrm{t}_0 \to \mathrm{t} = t_1$

(3) $\rho(\mathrm{depth}) = n$, $\rho[\mathrm{depth} = \mathrm{n} + 1] = \rho'$

(4) $\rho' \vdash$ LET newc(n + 1)$_{\#\mathrm{d}}$ IN E :: t

---

(0) $(\lambda T_1\ \mathrm{IN}\ E) :> \mathrm{t}_1 \Rightarrow \mathrm{closure}(\rho', \mathrm{d}, \mathrm{E})$.

d is the parameter declaration, t_0 is the parameter type, e_0 is the argument value, t is the result type, and t_1 is the type of λ-exp.
The expression T_1 in the λ roughly gives the type of the entire λ-expression. Thus

(B) λi: int → int IN $i+1$

has $T_1 = (i$: int → int), and its type (called t_1) is int → int. The value of T_1 is called t_1'; it differs from t_1 in that the declaration i: int has been reduced to its type int. This is done by (→I1), which accepts a T_1 which evaluates to something of the form $d \to i$, and computes first t_0 as typeOf d (using $\rightsquigarrow e$ to evaluate typeOf), and then t_1 as $t_0 \to t$. The $\rightsquigarrow e$ rule for typeOf just decomposes the declaration to the primitive form N: t, and then strips off the N to return t. The cases for dependent and inferred products (lines 2 and 5) are discussed later.

The idea of (→I4) is that if we can show that (1) type-checks without any knowledge of the argument values, depending only on their types, then whenever the closure is applied to an expression with type t, the resulting (1) will surely type-check. This is the essence of static type-checking: the definition of a function can be checked independently of any application, and then only the argument type need be checked on each application. (→I4) is true if we can show that

$$\text{LET newc}(n+1)_{\#d} \text{ IN } E \tag{2}$$

has the result type t, where newc$(n+1)$ is a constant, about which we know nothing except that its type is d. In other words, newc$(n+1)$ is a binding for the names in d, in which each name has the type assigned to it by d. Here n is the depth of nesting of λ-expressions. It is straightforward to show that newc$(n+1)$ does not appear in ρ, and therefore does not appear in t either. This ensures that the proof that (2) has type t does not depend on the values of the arguments.

For our example (B), we have

$$\text{LET newc}(1)_{\#i:\text{int}} \text{ IN } i+1 \tag{3}$$

which must have type int. To show this we need the base case of :E, the rule for LET,

$$:\text{E} \quad \frac{(3)\ \text{B} :: (N: t_0), \text{rhs B} \Rightarrow e_0, \rho[\text{N} = \text{t}_0 \sim \text{e}_0] \vdash \text{E}:> t \Rightarrow e}{(0)\ \text{LET } B \text{ IN } E:> \text{t} \Rightarrow \text{e}}$$

Using this, (3) has type int if

$\rho[i = \text{int} \sim \text{rhs!newc}(1)] \vdash i + 1$

has type int. Since $i + 1$ is sugar for plus$[i, 1]$, its type is given by the result type of plus (according to $\rightarrow$E1), provided that $[i, 1]$ has the argument type of plus. Since

$\text{plus} :: \text{int} \times \text{int} \rightarrow \text{int}$

we have the desired result if $[i, 1] :: \text{int} \times \text{int}$. Using $\times$I this is true if $i :: \text{int}$ and $1 :: \text{int}$. According to NE, the former is true if $\rho(i) \approx \text{int} \sim e_0$. But in fact $\rho(i) = \text{int} \sim \text{rhs!newc}(1)$, so this is established. Similarly, the initial environment tells us that $\rho(1) = \text{int} \sim 1$.

We can write this argument more formally as follows:

(B1) $\rho \vdash \text{LET newc}(1)_{\#i} : \text{int IN } i + 1 :: \text{int}$	2, : E
(B2) $\rho_1 \vdash i + 1 :: \text{int}$,	3, $\rightarrow$ E
where $\rho_1 = \rho[i = \text{int} \sim \text{rhs!newc}(1)]$	
(B3) $\rho_1 \vdash \text{plus} :: t \rightarrow \text{int}, [i, 1] :: t$	4, 5
(B4) $\rho_1 \vdash \text{plus} :: \text{int} \times \text{int} \rightarrow \text{int}$	7, NE
(B5) $\rho_1 \vdash [i, 1] :: \text{int} \times \text{int}$	5, $\times$E
(B6) $\rho_1 \vdash i :: \text{int}, 1 :: \text{int}$	7, NE
(B7) $\rho_1(i) \approx \text{int} \sim e_0, \rho_1(1) \approx \text{int} \sim 1,$	
$\rho_1(\text{plus}) \approx \text{int} \times \text{int} \rightarrow \text{int} \sim \text{primitive(plus)}$	inspection.

We now consider the non-dependent case of application, and return to λ-expressions with dependent types in the next section,

$\rightarrow$E (1) $F :: t_0 \rightarrow t, \text{coerce}(E_0, t_0) :: t_0 \Rightarrow e_0$

(3) $\{\text{f!e}_0 \rightsquigarrow e \text{ else f!e}_0 = e\}$

(0) $F\,E_0 :> t \Rightarrow e$

The type-checking is done by $\rightarrow$E1, which simply checks that the argument E_0 can be coerced to the parameter type t_0 of the function. The coercion is done by typeE; line (1) of this rule says that if E has type t, then it can be coerced to type t simply by evaluating it. Line (2) says that if E has type t', and there is a coercion function coerce$F(t',t)$, then E can be coerced to t by applying the function. The coercion function is computed by $\rightsquigarrow h$, which has two parts. Lines (1–5) compute coercions for constructed types from those for simpler types: a product can be coerced by coercing its first and second parts, a function by composing it with a coercion from the desired argument type and a coercion to the desired result type, and a declaration by coercing the value part. Lines (6–8) give coercion

rules for particular types, which are discussed in connection with these types: inferred products, bindings, and extended types. Coercions are not a fundamental part of the language, but they are a great convenience to the programmer in handling the inheritance relations among abstract types.

$\rightarrow E3$ tries to use the $\rightsquigarrow$ rules for evaluating applications to obtain the value of f when applied to the argument value e_0. If no $\rightsquigarrow$ rule is applicable, the value is just $f!e_0$, i.e., a more complex symbolic value. The $\rightsquigarrow$ rules have two main cases, depending on whether f is a primitive or a closure. For f an arbitrary primitive f_0 we use the main $\rightsquigarrow$ rule,

$$\rightsquigarrow \quad \frac{\text{for each } \langle \text{arg, result} \rangle \text{ pair in each primitive } f_0}{(0)\ f_0!e_0 \rightsquigarrow e}$$

Because of the type-check, this will succeed for a properly constructed primitive unless e_0 is a symbolic value, i.e., contains a newc constant or a fix.

Thus the $\rightsquigarrow$ rules can be thought of as an evaluation mechanism for primitives which is programmed entirely outside the language, as is appropriate for functions which are primitive in the language. In its simplest form, as suggested by the $\rightsquigarrow$ rule above, there is one rule for each primitive and each argument value, which gives the result of applying that primitive to that value. More compact and powerful rules are also possible, however, as $\rightsquigarrow a-c$ illustrate.

Note that the soundness of the type system depends on consistency between the types of a primitive (as expressed in rules like $\times \mathrm{E} a-b$), and the $\rightsquigarrow$ rules for that primitive ($\rightsquigarrow a-b$ for fst and snd). For each primitive, a proof is required that the $\rightsquigarrow$ rules give a result for every argument of the proper type, and that the result is of the proper type.

If f is closure(ρ_0, d, E), $\rightsquigarrow d$ first computes typeOf d, which is the type that the argument e_0 must have. Then it evaluates the closure body E in the closure environment ρ_0 augmented by the binding $d \sim e_0$. Note the parallel with $\rightarrow$I4, which is identical except that the unknown argument binding $\text{newc}_{\#d}$ replaces the actual argument binding $d \sim e_0$. The success of the type-check made by $\rightarrow$I4 when f was constructed ensures that the LET in $\rightsquigarrow d$ will type-check.

The remaining $\rightsquigarrow$ rules evaluate the primitives typeOf (discussed above), $\sim$ (Section 5.2.4), fixtype (Section 5.2.5), coerceF (discussed above), shrinkF (Section 5.2.4), and xtd^{-1} (Section 5.2.4).

If f is neither a primitive nor a closure, it must be a symbolic value. In this case there is not enough information to evaluate the application, and $\rightarrow$E3 leaves it in the form $f!e_0$. There is no hope for simplifying this in any larger context.

5.2.3. *Dependent functions.* We now return to the function rule, and consider the case in which the λ-expression has a dependent type,

(2) $T_1 \Rightarrow t_1$, $t_1 \approx d \rhd f$, f!newc(n + 1) $\rightsquigarrow t$,
(3) ρ(depth) = n, ρ[depth = n + 1] = ρ'
(4) $\rho' \vdash$ LET newc(n + 1)$_{\#d}$ IN E :: t
(0) (λ T_1 IN E) :> $t_1 \Rightarrow$ closure(ρ', d, E)

The only difference is that →I2 applies instead of →I1; it deals with a function whose result type depends on the argument value, such as the swap function defined earlier by

(C) *swap*: ~ $\lambda(t_1$: type × t_2: type) →» ($t_1 \times t_2 \rightarrow t_2 \times t_1$) IN
$\lambda(x_1: t_1 \times x_2: t_2) \rightarrow t_2 \times t_1$ IN [x_2, x_1]

The type expression for the type of *swap* (following the first λ) is sugar for

(t_1: type × t_2: type) ▷ (λ B': (t_1: type × t_2: type) → type
IN LET B' IN ($t_1 \times t_2 \rightarrow t_2 \times t_1$))

The operator ▷ is very much like →, but where → has the simple type
type × type→type
▷ has the more complex type
d: type ×× f: (d→type)→type
Thus the type of *swap* is

(t_1: type × t_2: type) ▷ (4)
closure(ρ, B': (t_1: type × t_2: type), LET B' IN $t_1 \times t_2 \rightarrow t_2 \times t_1$).

In this case the parameter type of *swap* is just (t_1: type × t_2: type); we do not use typeOf to replace it with type × type. This would be pointless, since the names t_1 and t_2 would remain buried in the closure, and to define equality of closures by the α-conversion rule of the λ-calculus would take us afield to no good purpose. Furthermore, if elsewhere in the program there is another type expression which is supposed to denote the type of *swap*, it must also have →» as its main operator, and a declaration with names corresponding to t_1 and t_2. This is in contrast with the situation for a non-dependent functon type, which can be written without any names. The effect of leaving the names in, and not providing α-conversion between closures, is that two dependent function types must use the same names for the parameters if they are to be equal. (Note, in a more recent version of Pebble, incorporating many changes, we provide an equality for closures which is true when they are α-convertible.)

We do, however, need to compute an intended result type against which to compare the type of (1). This is done by applying the closure in (4) to newc(1); note that this new constant is the same here and in the instantiation of →I4. In this example, this application yields

rhs!fst!newc(1) × rhs!snd!newc(1)→rhs!snd!newc(1) × rhs!fst!newc(1))

which we call t.

The body is typechecked as before using →I4. It goes like this

(C1) $\rho \vdash$ LET newc(1)$_{\#\, t_1:\text{type} \times t_2:\text{type}}$ IN
$\lambda\, x_1{:}\, t_1 \times x_2{:}\, t_2 \to t_2 \times t_1$ IN $[x_2, x_2]$ :: 2, :E
rhs!fst!newc(1)×rhs!snd!newc(1)→rhs!snd!newc(1)×rhs!fst!newc(1))

(C2) $\rho_1 \vdash \lambda(x_1{:}\, t_1 \times x_2{:}\, t_2) \to t_2 \times t_1$ IN $[x_2, x_1]$ :: equality, 3, →I
rhs!fst!newc(1)×rhs!snd!newc(1)→rhs!snd!newc(1)×rhs!fst!newc(1)),

where $\rho_1 = \rho[t_1 = \text{type} \sim \text{rhs!fst!newc(1)}, t_2 = \text{type} \sim \text{rhs!snd!newc(1)}]$,

(C3) $\rho_1 \vdash$ LET newc(2)$_{\#\, x_1:\, \text{rhs!fst!newc(1)} \times x_2:\, \text{rhs!snd!newc(1)}}$ IN$[x_2, x_1]$ :: 4, :E
rhs!snd!newc(1) × rhs!fst!newc(1))

(C4) $\rho_2 \vdash [x_2, x_1]$ :: rhs!snd!newc(1) × rhs!fst!newc(1)), 5, × E

where $\rho_2 = \rho_1[x_1 = \text{rhs!fst!newc(1)} \sim \text{rhs!fst!newc(2)},$
$x_2{:}\text{rhs!snd!newc(1)} \sim \text{rhs!snd!newc(2))}]$,

(C5) $\rho_2 \vdash x_2$ :: rhs!snd!newc(1), $\rho_2 \vdash x_1$ :: rhs!fst!newc(1) 6, NE

(C6) $\rho_2(x_2) \approx \text{rhs!snd!newc(1)} \sim e_{02}$, $\rho_2(x_1) \approx \text{rhs!fst!newc(1)} \sim e_{01}$
inspection.

Observe that we carry symbolic forms (e.g., rhs!snd!newc(1)) of the values of the arguments for functions whose bodies are being type-checked. In simple examples such as (A) and (B), these values are never needed, but in a polymorphic function like *swap* they appear as the types of inner functions. Validity of the proof rests on the fact that two identical symbolic values always denote the same value. This in turn is maintained by the applicative nature of our system; the fact that we generate a different newc(n) constant for each nested λ-expression where n is the depth of nesting maintained by the depth component of ρ, and the fact that if $\rho(\text{depth}) = n$, newc(n') with $n' > n$ never appears in ρ.

A function with a dependent type $d \rhd f$ is applied very much like an ordinary function,

→E (2) $F :: d \rhd f_1, f_{1\, \#\, d \to \text{type}}(d \sim E_0) \Rightarrow t, \text{rhs}(d \sim E_0) \Rightarrow e_0$

(3) $\{f!\, e_0 \rightsquigarrow e$ **else** $f!e = e\}$

(0) $F\, E_0 {:}> t \Rightarrow e$

The only difference is that →E2 is used for the type computation instead of →E1. This line computes the result type of the application by applying f_1 to the argument binding $d \sim E_0$; in evaluating this binding E_0 is coerced to the argument type typeOf d, which we call t_0. It is exactly parallel to →I2, which computes the (symbolic) result type of applying the function to the unknown argument binding newc$_{\#d}$. We apply f_1 to $d \sim E_0$ rather than to E_0 because typeIa, which constructs $d \triangleright f$, expects a binding as the argument of f_1. The reason for this is that in →E2 we do not have an expected type for E_0, but we do have a declaration d to which it can be bound. It is the evaluation of the binding $d \sim E_0$ that coerces the argument; there is no need for the explicit coercion of →E1.

5.2.4. *Bindings and declarations.* The main rule for binding shows how to use a binding in a LET to modify the environment in which a subexpression is evaluated (:E). A binding is constructed by the primitive function ~, defined by $\rightsquigarrow f$. The tricky case of recursive bindings (:Ia and NI2) is discussed in Section 5.2.5.

The type of ~ is given in Table VII; it is the value of

d: type ×× typeOf $d \twoheadrightarrow d$.

Thus it takes a declaration d, and a value whose type is typeOf d, and produces a binding of type d. It is defined by $\rightsquigarrow$f, which has four cases, depending on the form of the declaration value.

In evaluating $D \sim E$, if the declaration is void, typeOf D is void so that E must have type void also, and the result is nil. If D is $N: t$, it must be possible to coerce E to type typeOf D, which is t. If this yields e, the result is the binding value $N \sim e$; see Section 5.2.2 for a discussion of coercion. These are the base cases. If the declaration is $d_1 \times d_2$, E must have type typeOf $d_1 \times$ typeOf d_2, and the result is the value of $[d_1 \sim$ fst E, $d_2 \sim$ snd $E]$. Thus

i: int × x: real ~ [3, 3.14]

evaluates just like

[i: int ~ fst[3, 3.14], x: real ~ snd[3, 3.14]]

namely to $[i \sim 3, x \sim 3.14]$. All three of these cases yield d as the type of the binding.

The rule for a dependent declaration is more complicated. It is based on the idea that in the context of a binding, $d_0 = d_1 \diamond f_2$ can be converted to $d_1 \times d_2$ by applying f_2 to fst E to obtain d_2. The binding then has the type and value of $d_1 \times d_2 \sim E$. Thus

t: type ×× x: t ~ [int, 3]

has type t: type × x: int and evaluates to $[t \sim$ int, $x \sim 3]$. In this case the type of the binding is not d_0. but the simpler cross type $d = d_1 \times d_2$.

This idea is implemented by ⇝e4 and ⇝f3. The former computes typeOf $d_1 \diamond f_2$ as $t_1 \diamond f_2'$, where t_1 is typeOf d_1, just as for an ordinary product, and f_2' is the composition $f_2 \circ$ typeOf. In the rule, the composition is written out as a λ-expression. The shrinkF clause checks that D is really a declaration. To evaluate a binding to $d_1 \diamond f_2$, ⇝f3 applies f_2 to fst E to compute d_2, and then proceeds as for $d_1 \times d_2$.

For an inferred product $d_1 \otimes f_2$, ⇝e5 computes typeOf D as $t_1 \otimes f_2'$, where t_1 is typeOf d_1 as before, and f_2' is $f_2 \circ$ rhs, since f_2 infers a binding (of type fstt d_1) and f_2' should infer the rhs of that binding (of type fstt typeOf d_1). To evaluate a binding to $d_1 \otimes f_2$, just evaluate a binding to d_1; the argument has already been coerced to have the proper form.

The rule for LET B IN E has exactly the same cases,

$$
\begin{array}{ll}
:\mathrm{E} & (1)\quad \mathrm{B} :: \text{void}, \qquad \mathrm{E} :> t \Rightarrow e \\
& (2)\ \textbf{or}\ \mathrm{B} :: (N\colon t_0),\ \text{rhs B} \Rightarrow e_0, \qquad \rho[\mathrm{N} = \mathrm{t}_0 \sim \mathrm{e}_0] \vdash \mathrm{E} :> t \Rightarrow e \\
& (3)\ \textbf{or}\ \mathrm{B} :: d_1 \times d_2,\ \text{snd B} \Rightarrow b_2,\ \text{LET fst B IN LET } \mathrm{b}_{2\,\#\mathrm{d}_2} \text{ IN E} :> t \Rightarrow e \\
& (4)\ \textbf{or}\ \mathrm{B} :: d_1 \diamond f,\ \mathrm{f}_{\#\mathrm{d}_1 \to \mathrm{type}}(\text{fst B}) \Rightarrow d_2,\ \text{LET } \mathrm{b}_{\#\mathrm{d}_1 \times \mathrm{d}_2} \text{ IN E} :> t \Rightarrow e \\
\hline
& (0)\ \text{LET } B \text{ IN } E :> \mathrm{t} \Rightarrow \mathrm{e}
\end{array}
$$

If B has type void, the result is E in the current environment. If B has type N: t_0, the result is E in an environment modified so that N has type t_0 and value obtained by evaluating rhs B. Thus

LET i: int ~ 3 IN $i+4$

has the same type and value that $i+4$ has in an environment where i: int ~ 3, namely type int and value 7.

If B has a cross type, the result is the same as that of a nested LET which first adds fst B to the environment and then adds snd B. The rule evaluates snd B separately; if it said

LET fst B IN LET snd B IN E

the value of snd B would be affected by the bindings in fst B.

Finally, if B has a dependent type, that type is reduced to an ordinary cross type $d_1 \times d_2$, and the result is the same as LET B' IN E, where $B' = b_{\#d_1 \times d_2}$ has the same value as B, but an ordinary cross type. The last case will never arise in a LET with an explicit binding expression for B, since :I will always compute a cross type for such a B. However, when type-checking a function such as

λt: type $\times\times$ x: $t \to$ int IN E

→I4 requires a proof of

LET newc$_{\#d_1 \diamond f}$ IN E :: int

where $d_1 \diamond f$ is the value of t: type $\times\times$ x: t. :E4 reduces this to

LET newc $_{\#\, t:\, \text{type} \times x\ \text{fst!newc}}$ IN E :: int

$$\text{:Ea} \qquad \frac{\text{(a1) B} :: d \Rightarrow b,\ [\ ] \rightarrow \text{LET b}_{\#\, d}\ \text{IN E} :: t \Rightarrow e}{\text{(a0) IMPORT } B \text{ IN } E :> \text{t} \Rightarrow \text{e}}.$$

The IMPORT construct has a very simple rule, :Ea. This says that to evaluate IMPORT B IN E, evaluate E in an environment which contains *only* the binding of B.

There is a special coercion rule $\rightsquigarrow h7$ for bindings, which says that a binding B can be *shrunk* to a binding of type d. Shrinking is defined by $\rightsquigarrow i$, which calculates a function f that shrinks d' to d. It succeeds if for every simple declaration N: t of which d is composed, $B\$N$ has type t. The shrinking works by using N: $t \sim B\$N$ as the value corresponding to N: t, and putting these simple bindings together according to the structure of d. The motivation for shrinking is to allow extra elements to be dropped from a binding; see Section 2.3 for examples.

5.2.5. *Recursion.* Recursion is handled by a fixed point constructor. If F is an n-tuple of functions F_i with types $d \rightarrow d_i$, and $d_1 \times \cdots \times d_n = d$, then FIX F has type d and is the fixed point of F; i.e., $F(\text{FIX } F) = (\text{FIX } F)$. The novelty is in the treatment of mutual recursion: d may declare any number of names, and correspondingly FIX F binds all these names. The following sugar is convenient for constructing F:

REC $D_1 \sim E_1, \ldots, D_n \sim E_n$ for FIX($(\lambda B'$: $D_1 \times \cdots \times D_n$ IN LET B' IN $D_1 \sim E_1$

, ...,

$\lambda B'$: $D_1 \times \cdots \times D_n$ IN LET B' IN $D_n \sim E_n$)).

For example,

REC

g: (int $\rightarrow$ int) $\sim \lambda\, x$: int $\rightarrow$ int IN IF $x = 0$ THEN 1 ELSE $x * h(x/2)$.

h: (int $\rightarrow$ int) $\sim \lambda\, y$: int $\rightarrow$ int IN IF $y < 2$ THEN 0 ELSE $g(y-2)$

has type

g: (int $\rightarrow$ int) $\times$ h: (int $\rightarrow$ int).

Its value is a binding for g and h in which their values are the closures we would expect, with an environment ρ_{gh} that contains suitable recursive bindings for g and h. We shall soon see how this value is obtained, but for the moment let us just look at it:

[$g \sim$ closure(ρ_{gh}, x: int, IF $x = 0$ THEN 1 ELSE $x * h(x/2)$).

$h \sim$ closure(ρ_{gh}, y: int, IF $y < 2$ THEN 0 ELSE $g(y-2)$)]

where

$\rho_{gh} = \rho[g: (\text{int} \to \text{int}) \sim \text{rhs!fst!fix}(f,f), h: (\text{int} \to \text{int}) \sim \text{rhs!snd!fix}(f,f)]$,

where

$f = [\text{closure}(\rho, B': d, \text{LET } B' \text{ IN } g: (\text{int} \to \text{int}) \sim \lambda\, x: \text{int} \to \text{int IN}$
$\text{IF } x = 0 \text{ THEN 1 ELSE } x * h(x/2),$
$\text{closure}(\rho, B': d, \text{LET } B' \text{ IN } h: (\text{int} \to \text{int}) \sim \lambda\, y: \text{int} \to \text{int IN}$
$\text{IF } y < 2 \text{ THEN 0 ELSE } g(y-2)]$,

where

$d = g: (\text{int} \to \text{int}) \times h: (\text{int} \to \text{int})$

The fix values inside ρ_{gh} are what capture the infinite value of this recursive binding in our operational semantics. Of course, if g is looked up in ρ_{gh} (as it will be, for example, when we compute $h(3)$), we do not want to obtain rhs!fst!fix(f,f) as its value; rather, we want closure$(\rho_{gh}, x: \text{int}, \ldots)$. To get this we *unroll* the fix value; that is, we replace fix(f',f) by $f'(\text{fix}(f',f))$, which evaluates to a closure. This unrolling is done by the $\rightsquigarrow\!\!\!\rightarrow$ rule, which also deals with the possibility that there may be an operator such as rhs outside,

$$\rightsquigarrow\!\!\!\rightarrow \quad \frac{\begin{array}{l}(1)\ \text{e} \approx \text{w!e}_1,\ \text{e}_1 \rightsquigarrow\!\!\!\rightarrow e_2,\ \{\text{w!e}_2 \rightsquigarrow\!\!\!\rightarrow e' \text{ or } \text{w!e}_2 \rightsquigarrow e'\} \\ (2)\ \textbf{or}\ \text{e} \approx \text{w!fix}(f'',f),\ \{\text{w!f}' \rightsquigarrow f'',\ \text{fix}(\text{f}'',\text{f}) = e'\} \\ (3)\ \textbf{or}\ \text{f}' \approx \text{closure}(x),\ \text{f}'\text{!fix}(\text{f}',\text{f}) \rightsquigarrow e'',\ \text{w!e}'' \rightsquigarrow e'\}\end{array}}{(0)\ e \rightsquigarrow\!\!\!\rightarrow e'}$$

This rule unrolls rhs!fst!fix(f,f) by first computing

fst!fix$(f,f) \rightsquigarrow\!\!\!\rightarrow$ fix(f',f),

where f' is the value of fst!f (the functional for g), using $\rightsquigarrow\!\!\!\rightarrow$ 2, and then

fix$(f',f) \rightsquigarrow\!\!\!\rightarrow g \sim \text{closure}(\rho_{gh}, x: \text{int}, \ldots)$

using $\rightsquigarrow\!\!\!\rightarrow$ 3 and $\to$E, and finally simplifying rhs!fix(f',f) to closure$(\rho_{gh}, x: \text{int}, \ldots)$ using $\rightsquigarrow\!\!\!\rightarrow$ 1 and $\rightsquigarrow$c. Thus, each time g or h is looked up in ρ_{gh}, the NI and $\rightsquigarrow\!\!\!\rightarrow$ rules unroll the fix once, which is just enough to keep the computation going.

For the persistent reader, we now present in detail the evaluation of a simple recursive binding with one identifier, and an application of the resulting function. Since some of the expressions and values are rather long, we introduce names for them as we go. First the recursive binding:

(D) REC P: (int $\to$ int) $\sim \lambda\, n$: int $\to$ int IN
IF $n < 2$ THEN n ELSE $P(n-2)$

We can write this more compactly as

REC $DP \sim L$

where

$$\mathrm{DP} = P\colon (\mathrm{int} \to \mathrm{int}),$$

$$L = \lambda\, n\colon \mathrm{int} \to \mathrm{int}\ \mathrm{IN}\ \mathrm{EXP},$$

$$\mathrm{EXP} = (\mathrm{IF}\ n < 2\ \mathrm{THEN}\ n\ \mathrm{ELSE}\ P(n-2)).$$

The table below is a proof that the value of (D) is

$P\colon (\mathrm{int} \to \mathrm{int}) \sim \mathrm{closure}(\rho_{fp}, n\colon \mathrm{int}, \mathrm{EXP})$

It has been abbreviated by omitting the # types on values which are used as expressions. The evaluation goes like this. First we construct the λ-expression for the functional whose fixed point f we need (D3) and evaluate it to obtain a closure (D4). Then, according to :Ia2, we embed f in $\mathrm{fix}(f,f)$ and unroll it. This requires applying f to the fix (D5), which gives rise to a double LET (D6), one from the application and the other from the definition of the functional. After both LETs have their effect on the environment, we have ρ_{fp}, which contains the necessary fix value for P ((D7)-(D10)). Now evaluating the λ to obtain a closure value for P that contains ρ_{fp} is easy (D12)-(D13),

(D1)	$\rho \vdash \mathrm{REC}\ DP \sim L :: dp \Rightarrow bp$	:I, Ia, 3, 5
(D1a)	$\rho \vdash DP \Rightarrow dp$	typelc, 2
(D2)	$\rho \vdash P\colon \mathrm{int} \to \mathrm{int} = dp$	definition
(D3)	$\rho \vdash (\lambda B'\colon DP \to DP\ \mathrm{IN\ LET}\ B'\ \mathrm{IN}\ dp \sim L) \Rightarrow f$	$\to$I, 4
(D4)	$\mathrm{closure}(\rho, B'\colon dp, \mathrm{LET}\ B'\ \mathrm{IN}\ dp \sim L) = f$	definition
(D5)	$\rho \vdash f(\mathrm{fix}(f,f)) \Rightarrow bp$	$\to$E, 6
(D6)	$\rho \vdash B'\colon dp \sim \mathrm{fix}(f,f) \Rightarrow bf,$ $\rho \vdash \mathrm{LET}\ bf\ \mathrm{IN\ LET}\ B'\ \mathrm{IN}\ dp \sim L \Rightarrow bp$	#, $\rightsquigarrow f$, 7,:E, 8
(D7)	$B'\colon dp \sim \mathrm{fix}(f,f) = bf$	definition
(D8)	$\rho_f \vdash \mathrm{LET}\ B'\ \mathrm{IN}\ dp \sim L \Rightarrow bp,$	:E, 9, 10

where $\rho_f = \rho[B'\colon dp \sim \mathrm{fix}(f,f)]$,

(D9)	$\rho_f \vdash \mathrm{rhs}\ B' \Rightarrow \mathrm{rhs!fix}(f,f)$	$\to$E, NI
(D10)	$\rho_{fp} \vdash dp \sim L \Rightarrow bp,$	$\rightsquigarrow$f, 11, 12

where $\rho_{fp} = \rho_f[P\colon (\mathrm{int} \to \mathrm{int}) \sim \mathrm{rhs!fix}(f,f)]$,

(D11)	$\rho_{fp} \vdash L \Rightarrow \mathrm{closure}(\rho_{fp}, n\colon \mathrm{int}, \mathrm{EXP})$	$\to$I
(D12)	$P \sim \mathrm{closure}(\rho_{fp}, n\colon \mathrm{int}, \mathrm{EXP}) = bp$	definition.

Note that this evaluation does not depend on having λ-expressions for

the values of the recursively bound names. It will work fine for ordinary expressions, such as

REC i: int $\sim j+1, j$: int ~ 0

which binds $i: \sim 1$ and $j: \sim 0$. However, it may not terminate. For instance, consider

REC (i: int $\times j$: int) $\sim [j+1, i]$.

In fact,

REC (i: int $\times j$: int) $\sim [j+1, 0]$

will also fail to terminate, because the rules insist on evaluating $[j+1, 0]$ in order to obtain the 0 which is the value of j. REC also fails if type-checking requires the values of any of the recursively defined names, as in

REC t: type $\sim$ int, g: int $\to$ int $\sim \lambda\, i$: int $\to$ int IN LET x: $t \sim i+1$ IN $x * x$

because in type-checking the f function we only have a newc value for t, not its actual value int.

Now we look at an application of P:

(E) LET (REC P: (int $\to$ int) $\sim \lambda\, n$: int $\to$ int IN
IF $n < 2$ THEN n ELSE $P(n-2)$)
IN $P(3)$

This has type int and value 1, as we see in the proof which follows. First we get organized to do the application with the proper recursive value for P (E1)-(E2). The application becomes a LET after P and 3 are evaluated (E3)-(E5). This results in an environment ρ_{n3} in which $n \sim 3$, so we need to evaluate $P(n-2)$ (E6)-(E7). Looking up P we find a value which can be unrolled (E8)-(E9) to obtain the recursive value closure(ρ_{fp}, n: int, EXP) again (E10)-(E11). Since $n-2 \Rightarrow 1$ (E12), we get the answer without any more recursion (E13)-(E15).

(E1)	$\rho \vdash$ LET REC $DP \sim L$ IN $P(3)$ :: int $\Rightarrow$ 1	:E, D, 2
(E2)	$\rho_p \vdash P(3)$ :: int $\Rightarrow$ 1,	$\to$ E, 3, 4, 5

where $\rho_p = \rho[P = (\text{int} \to \text{int}) \sim \text{closure}(\rho_{fp}, n\text{: int, EXP})]$,

(E3)	$\rho_p \vdash P$ :: (int $\to$ int) $\Rightarrow$ closure(ρ_{fp}, n: int, EXP)	NI
(E4)	$\rho_p \vdash n$: int $\sim 3 \Rightarrow n \sim 3$	$\rightsquigarrow$ f, intI
(E5)	$\rho_{fp} \vdash$ LET $n \sim 3$ IN EXP :: int $\Rightarrow$ 1	:E, 6
(E6)	$\rho_{n3} \vdash$ IF $n < 2$ THEN n ELSE $P(n-2)$ :: int $\Rightarrow$ 1	boolE, 7

where $\rho_{n3} = \rho_{fp}[n = \text{int} \sim 3]$

(E7) $\rho_{n3} \vdash P(n-2) :: \text{int} \Rightarrow 1$ → E, 8, 12, 13
(E8) $\rho_{n3} \vdash P :: (\text{int} \to \text{int}) \Rightarrow \text{closure}(\rho_{fp}, n\text{: int, EXP})$ NI, 9
(E9) $\text{rhs!fix}(f, f) \rightsquigarrow\!\!\!\!\rightarrow \text{closure}(\rho_{fp}, n\text{: int, EXP})$ $\rightsquigarrow\!\!\!\!\rightarrow$, 11
(E11) $\text{rhs!}bp \rightsquigarrow \text{closure}(\rho_{fp}, n\text{: int, EXP})$ $\rightsquigarrow b$
(E12) $\rho_{n3} \vdash n-2 :: \text{int} \Rightarrow 1$ NI, → E, $\rightsquigarrow$
(E13) $\rho_{fp} \vdash \text{LET } n \sim 1 \text{ IN EXP} :: \text{int} \Rightarrow 1$:E, 14
(E14) $\rho_{n1} \vdash \text{IF } n < 2 \text{ THEN } n \text{ ELSE } P(n-2) :: \text{int} \Rightarrow 1,$ boolE, 15

where $\rho_{n1} = \rho_{fp}[n = \text{int} \sim 1]$,

(E15) $\rho_{n1} \vdash n :: \text{int} \Rightarrow 1.$ NI.

It should be clear to anyone who has followed us this far that we have given a standard operational treatment of recursion. There is some technical interest in the way the fix is unrolled, and in the handling of mutual recursion.

5.2.6. *Inferring types.* The inference rules give a way of computing a type for any expression. In some cases, however, an expression may have additional types. In particular, this happens with types of the form $d \diamond f$ and typeOf!$(d \diamond f)$, because pairs with these dependent types also have ordinary cross types, which are the ones computed by the inference rules. To express this fact, there is an additional inference rule :: which tells how to infer types that are not computed by the rest of the rules,

$$\frac{\begin{array}{l}(1)\ \text{E:} > \text{t} \\ (2)\ \textbf{or}\ \{\text{t} \approx t_1 \diamond f, \text{E} :: t_1 \times t_2\ \textbf{or}\ \text{t} \approx t_1 \times t_2, \text{E} :: t_1 \diamond f\}, \\ \qquad \lceil_{\#\,t_1 \to \text{type}}(\text{fst E}) \Rightarrow \text{t}_2 \\ (3)\ \textbf{or}\ \text{E} :: \text{t} \otimes f\ \textbf{or}\ \text{t} \approx t' \otimes f, \text{E} :: \text{t}' \\ (4)\ \textbf{or}\ \text{E} :: t', \{\text{t}' \approx \text{xt!}(d, (N \sim \text{t}, b))\ \textbf{or}\ \text{t} \approx \text{xt!}(d, N \sim \text{t}', b))\}\end{array}}{(0)\ E :: t}.$$

::1 says that the principal type of E is one of its types. ::2 turns $d \diamond f$ into $d \times t$ by applying f to fst E to compute t; then it checks that E has type $d \times t$. This is a reflection of the fact already discussed, that a pair may have many dependent types, as well as its "basic" cross type. This inference can go in either direction. ::3 says that if E has type $t \otimes f$, then it has type t. This follows from $\rightsquigarrow$gb which gives the only way of introducing a value with $\otimes$ type. This inference can also go in either direction. ::4 says that if E has an extended type xt(d, b), it also has the base type rhs fst b, and vice versa. This rule reflects the idea that extended types are a packaging mechanism for asociating a set of named functions and other values with a

type so that the whole package can be handled as a unit, but they do not introduce any new kinds of values, or provide any protection. The rule is sound because it is the only rule that applies to expressions with extended types.

5.3. *Execution*

The inference rules in Table VI tell us how to simultaneously type-check and evaluate a Pebble expression. With a few changes, however, we can turn them into rules which type-check an expression and produce *code* which can be executed to yield a value; these can reasonably be regarded as rules for a compiler. The code takes the form of a symbolic value consisting of the primitives of Table VII and newc(n), combined recursively using one of the forms

(e_1, e_2), $e_1!e_2$, N: e, $N \sim e$, cl([ ], e, n),

where N is a name and n is a natural number. The cl form is the result of evaluating a λ-expression; its intuitive meaning is that if values are supplied for newc(1), ..., newc(n), then e can be evaluated to yield a value without any newc's.

Two changes are required. The last two lines of the λ rule, $\rightarrow$I, become

$$\frac{(4)\ \rho' \vdash \text{LET newc(n + 1)}_{\#d}\ \text{IN E} :: \text{t} \Rightarrow e}{(0)\ (\lambda\, T_1\ \text{IN}\ E) :> \text{t} \Rightarrow \text{cl([ ], e, n + 1)}}$$

The $\rightsquigarrow d$ rule for applying a closure is replaced by a rule for applying a cl:

$$\frac{(\text{d1})\ \text{s} + (\text{n} \sim \text{e}_0) \vdash \text{e} \mapsto e'}{(\text{d0})\ \text{cl}(s, e, n)!\, e_0 \rightsquigarrow \text{e}'}$$

This rule makes use of a new function for executing a code expression e in the run-time environment s. We write $s \vdash e \mapsto e'$, meaning execute e in run-time environment s to yield e'. The rule above says that to apply a cl to e_0, add e_0 to the existing run-time environment and then execute e.

An s has the form $n_1 \sim e_1 + \cdots + n_k \sim e_k$; it supplies values for the arguments of procedures at levels $n_1, \ldots, n_k$ which were referred to symbolically as newc(n_1), ..., newc(n_k) in e. Two such objects which define disjoint sets of n_i can be combined in the obvious way to yield an s which defines the union of the sets.

The execution function is defined by the $\mapsto$ rule in Table VIII. Most of the lines just apply the rule recursively. Line (2) uses the $\rightsquigarrow$ rules to evaluate an application of a primitive or a cl. Line (7) evaluates newc(n) by looking up n in the environment s; if s is not rich enough, it leaves the newc(n) alone.

TABLE VIII

Run-Time Execution

(1)	$e \approx (e_1, e_2), s \vdash e_1 \mapsto e_1', s \vdash e_2 \mapsto e_2', (e_1', e_2') = e'$
(2)	or $e \approx e_1 ! e_2, s \vdash e_1 \mapsto e_1', s \vdash e_2 \mapsto e_2', e_1' ! e_2' \rightsquigarrow e'$
(3)	or $e \approx \text{let}(b, e_1, n), s \vdash b \mapsto b', s + (n \sim b') \vdash e_1 \mapsto e'$
(4)	or $e \approx N\colon t, s \vdash t \mapsto t', N\colon t' = e'$
(5)	or $e \approx N \sim e_1, s \vdash e_1 \mapsto e_1', N \sim e_1' = e'$
(6)	or $e \approx \text{cl}(s', e_1, n), \text{eqS}(s + s', s_1 + s_2), s_1 \vdash e_1 \mapsto e_1', \text{cl}(s_2, e_1', n) = e'$
(7)	or $e \approx \text{newc}(n), \{s(n) = e' \text{ else } e = e'\}$
(8)	else $e = e'$
	---
(0)	$s \vdash e \mapsto e'$
(1)	$s' = n \sim e, \text{lookup}(s, n, e)$
(2)	or $s' = s_1' + s_2', \text{includes}(s, s_1'), \text{includes}(s, s_2')$
	---
(0)	$\text{includes}(s, s')$
(1)	$s = n \sim e$
(2)	or $s = s_1 + s_2, \{\text{lookup}(s_1, n, e) \text{ or } \text{lookup}(s_2, n, e)\}$
	---
(0)	$\text{lookup}(s, n, e)$
(1)	$\text{includes}(s, s'), \text{includes}(s', s)$
	---
(0)	$\text{eqS}(s, s')$

Line (6) evaluates a cl in two steps. First, it augments the cl's saved environment by adding in the current environment. Then it non-deterministically splits the resulting s into two parts s_1 and s_2, and executes e_1 in s_1; this will supply values for some of the newc's in e_1, and do any applications which knowing these values allows. The result is a new piece of code e_1', which still needs the values in s_2, as well as a value for its argument newc(n), to be completely evaluated. We therefore bundle it back up into $\text{cl}(s_2, e_1', n)$. If s_1 is $(n \sim e)$, this corresponds to one step of β reduction in which e is substituted for newc(n). The non-determinism reflects the freedom of the implementation to do the substitution and partial evaluations in any order. We should prove, as a theorem, that the result of applying the closure does not depend on how these choices are made.

Note that during this execution types play no role; the type-checking is all done during the process of constructing the code. The converse is not true, however; during type-checking, when a type-returning function is applied, it may be necessary to execute the application in order to obtain a sufficiently reduced value. For example, consider

LET $id(t\colon \text{type} \rightarrow\!\!\!\!\rightarrow (t \rightarrow t)) \colon \sim (\lambda\, x\colon t \text{ IN } x)$ IN $id(\text{int})(3) + 1$

The type of *id* is

t: type $\triangleright$ cl([ ], $\rightarrow$!(rhs!newc(n), rhs!newc(n)), n)

and the result type of *id*(int) will be cl($\cdots$)!($t \sim$ int). In order to type-check the "+", this must be reduced to int by the $\rightsquigarrow$ and $\mapsto$ rules.

In general, however, it is not necessary or appropriate to apply a cl whenever the function part becomes known. Doing this is doing inline expansion of applications whenever possible, and leads to bulky, perhaps even infinite code. It is therefore appropriate to modify the uses of $\mapsto 2$ by using some heuristics to decide whether a cl should be applied. If e is a top-level symbolic value $e_1!e_2$, and $e_1 \mapsto$ cl($\cdots$), it should definitely be applied; this rule ensures that any cl application not nested inside another cl will be done. Otherwise the cl probably should not be applied, unless the body is fairly short, or this is the only application.

A related issue is the treatment of LET *B* IN *E*. The evaluation rules in Table VI always substitute for the names in the *B* wherever these names appear in *E*. This also is likely to increase the size of the result. An alternative is to treat LET more like λ, by changing :E2 to

or B :: (N: t_0), {rhs B $\Rightarrow e_0$, ρ[N = $t_0 \sim e_0$]$\vdash$E :: $t \Rightarrow e$
or (λN: t_0 IN E) :: $t_0 \rightarrow t \Rightarrow$ cl([ ], e, n), let(b, e, n) = e}

Later $\mapsto 3$ will evaluate the let, using the same heuristics as those for cl. The or in this rule allows a non-deterministic choice about expanding the LET, unless the second choice fails to type-check because the type-correctness of E depends on the value bound to N by B; this is likely to be true if t_0 is type, and may be true in other cases as well.

5.4. *Deterministic Evaluation*

As we mentioned in Section 5.1.2, it is possible to construct a deterministic evaluator from the inference rules. An experimental implementation of Pebble, without a parser, was made in Prolog by Glen Stone at Manchester University. A later one in ML, with a parser, was made at Edinburgh by Hugh Stabler, implementing this paper except for inferred types and extended types. Neither of these had pretensions to efficiency, but they checked out the semantics and uncovered one or two bugs.

6. Conclusion

We have presented both an informal and a formal treatment of the Pebble language, which adds to the type lambda calculus a systematic treatment of sets of labelled values, and an explicit form of polymorphism. Pebble can give a simple account of many constructs for programming in

the large, and we have demonstrated this with a number of examples. The language derives its power from its ability to maniplate large, structured objects without delving into their contents, and from the uniform use of λ abstraction for all its entities.

A number of areas are open for further work:

- Assignment, discussed briefly in Section 3.5.
- Exception-handling, as an abbreviation for returning a union result and testing for some of the cases,
- Concurrency. We do not have any ideas about how this is related to the rest of Pebble.
- A more mathematical semantics for the language (cf. Cardelli, 1986).
- Proof of the soundness of the type-checking, and an exploration of its limitations.

ACKNOWLEDGMENTS

We thank a number of people for helpful discussions over an extended period, particularly Luca Cardelli, Joseph Goguen, David MacQueen, Gordon Plotkin, Ed Satterthwaite, and Eric Schmidt. Valuable feedback on the ideas and their presentation was obtained from members of the IFIP Working Group 2.3. We are grateful to the referees for corrections and suggestions. Much of our work was supported by the Xerox Palo Alto Research Center. Rod Burstall also had support from the Science and Engineering Research Council, and he was enabled to complete this work by a British Petroleum Venture Research Fellowship and a SERC Senior Fellowship. We thank Eleanor Kerse for kindly typing the manuscript in Scribe format through several revisions and Oliver Schoett for Scribing the inference rules

RECEIVED October 16, 1985; ACCEPTED October 27, 1988

REFERENCES

AMADIO, R., AND LONGO, G. (1986), Type-free compiling of parametric types, *in* "IFIP Conference on Formal Description of Programming Languages, Ebberup, DK."

BAUER, F. L. *et al.* (1978), Towards a wide spectrum language to support program specification and program development, *SIGPLAN Notices* 13, 15–24.

BURSTALL, R. (1984), Programming with modules as typed functional programming, *in* "Proc. International Conference on Fifth Generation Computing Systems, ICOT, Tokyo."

BURSTALL, R., AND GOGUEN, J. (1977), Putting theories together to make specifications, *in* "5th Joint International Conference on Artificial Intelligence, Combridge, MA," pp. 1045–1058.

CARDELLI, L. (1984), A semantics of multiple inheritance, "Lecture Notes in Computer Science Vol. 173," pp. 51–68, Springer-Verlag, Berlin/New York.

CARDELLI, L. (1986), "A Polymorphic Lambda Calculus with Type: Type," Report 10, Digital Equipment Corp. Systems Research Center, Palo Alto, CA.

DEMERS, A., AND DONAHUE, J. (1980), Datatypes, parameters and type-checking, *in* "7th ACM Symposium on Principles of Programming Languages, Las Vegas," pp. 12–23.

GIRARD, J-Y. (1972), "Interpretation Fonctionelle et Elimination des Coupures dans l'Arithmetique d'Ordre Superieur," These de Doctorat d'etat, University of Paris.

GORDON, M., MILNER, R., AND WADSWORTH, C. (1979), Edinburgh LCF, "Lecture Notes in Computer Science Vol. 78," Springer-Verlag, Berlin/New York.

HARPER, R., MACQUEEN, D., AND MILNER, R. L (1986), Standard ML, Computer Science Department, Edinburgh University, Report ECS-LFCS-86-2.

LAMPSON, B., AND SCHMIDT, E. (1983), Practical use of a polymorphic applicative language, *in* "10th ACM Symposium on Principles of Programming Languages, Austin."

LAMPSON, B. W. (1983), "A Description of the Cedar Language," Report CSL-83-15, Xerox Palo Alto Research Center.

LANDIN, P. (1964), The next 700 programming languages, *Comm. ACM* 9, 157–166.

MACQUEEN, D., AND SETHI, R. (1982), A higher order polymorphic type system for applicative languages, *in* "Symposium on Lisp and Functional Programming, Pittsburgh, PA." pp. 243–252.

MACQUEEN, D., PLOTKIN, G., AND SETHI, R. (1984), An ideal model for recursive polymorphic types, *in* "11th ACM Symposium on Principles of Programming Languages, Salt Lake City."

MACQUEEN, D. (1984), "Modules for Standard ML (Draft)," Computer Science Department, Edinburgh University.

MARTIN-LOF, P. (1973), An intuitionistic theory of types: Predicative part, *in* "Logic Colloq. '73" (H. E. Rose and J. C. Shepherdson, Eds.), pp. 73–118, North-Holland, Amsterdam/New York.

MCCRACKEN, N. (1979), "An Investigation of a Programming Language with a Polymorphic Type Structure," Ph. D. thesis, Computer and Information Science, Syracuse University.

MILNER, R. (1978), A theory of type polymorphism in programming, *J. Comput. System Sci.* 17(3), 348–375.

MITCHELL, J., MAYBURY, W., AND SWEET, R. (1979), "Mesa Language Manual," Report CSL-79-3, Xerox Palo Alto Research Center.

PEPPER, P. (1979), "A study on Transformational Semantics," Dissertation, Fachbereich Mathematik, Technische Universität Munchen.

PLOTKIN, G. (1981), "A Structural Approach to Operational Semantics," Computer Science Department Report, Aarhus University.

PLOTKIN, G., AND MITCHELL, J. (1985), Abstract Types have Existential Type *in* "Twelfth International Conference on Principles of Programming Languages, New Orleans."

REYNOLDS, J. (1974), Towards a theory of type structure, "Lecture Notes in Computer Science Vol. 19," pp. 408–425, Springer-Verlag, Berlin/New York.

REYNOLDS, J. (1983), Types, abstraction and parametric polymorphism, *in Inform. Process.*, 83.

SCHMIDT, E. (1982), "Controlling Large Software Development in a Distributed Environment," Report CSL-82-7, Xerox Palo Alto Research Center.