1

# A Kernel Language for Modules and Abstract Data Types

R. Burstall and B. Lampson

September 1, 1984

**Systems Research Center**
130 Lytton Avenue
Palo Alto, California 94301

# A Kernel Language for Modules and Abstract Data Types

R. Burstall and B. Lampson

R. Burstall is at the Department of Computer Science, University of Edinburgh, King's Buildings, Mayfield Road, Edinburgh 9.
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 edited by G. Kahn, D. B. MacQueen, and G. Plotkin, #173 in the Springer-Verlag Lecture Notes in Computer Science, pages 1 to 50.

Authors' abstract:

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, and types as first-class 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.

R. Burstall
and B. Lampson

Capsule review:

Programming-language designers have invented a variety of language extensions and special notations to deal with several problems that arise in programming in the large. Some of the differences among such features in Ada, CLU, Euclid, Mesa, ML, Modula, Russell, SML, et al. are superficial; others are fundamental. Without a uniform semantic framework it is difficult to compare and evaluate these features, or to determine which choices are arbitrary and which are tightly constrained.

Pebble is a simpler language, intended for the precise description of language constructs. It is used to explain strongly typed module interconnection languages, abstract data types, and procedures that are parameterized with respect to the types of operands. It is based on the typed lambda calculus, extended to encompass the linking together of separately checked modules into a program. Bindings, declarations, and types -- as well as functions -- are all treated as first-class values; the type system includes dependent types.

This paper presents an informal overview of why the approach can be expected to work. But the precise definition of the features of existing languages in terms of Pebble is left as "an exercise for the reader."

The semantics of Pebble are presented both informally and formally. Representative cases are presented in great detail, to illustrate the workings of the formalism.

Jim Horning

Contents

## 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 too 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 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 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 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. 1979] 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. One of us (BL) 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; this work is described in an unpublished report.

### *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 idea 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:

implementation ⇄ value

interface ⇄ type

module ⇄ 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, though 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 (BL). 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 by 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.

*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 have to 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 only need to be handled 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. Another extension is needed because, whilst 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 ($x$~1, $y$~5, $z$~3). This leads us to the notion of a *binding*. To elucidate the notion of 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 give a precise semantics of Pebble 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. So far as we know it would be quite possible ~~possible~~ to give a satisfactory denotational semantics for Pebble, and we should be interested to learn of anyone attempting this task. 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. We give a fragment of such a translation in § 5.4.

*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, 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 each independently noted the usefulness of existentially quantified types for explaining data abstraction. We had already noticed 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".

The main difference of our approach from that using quantified types is that we take types as values and 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].

*Acknowledgements*

We would like to thank a number of people for helpful discussions over an extended period, particularly Jim Donahue, Joseph Goguen, David MacQueen, Gordon Plotkin, Ed Satterthwaite and Eric Schmidt. Valuable feedback on the ideas and their presentation was obtained from members of IFIP Working Group 2.3. Much of our work was supported by the Xerox Palo Alto Research Center. Rod Burstall also had support from the Science Research Council, and he was enabled to complete this work by a British Petroleum Venture Research Fellowship.

## 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;

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 the values and the formal syntax, given in § 4, when he is unclear about some point. Likewise the operational semantics given in § 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.

We start by describing the values, which we write in this font. 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, 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 ▸,
  - the type type which is the type of all types including itself, and
  - declarations, such as $x$: int, which are the types of bindings;
- applications: primitive functions applied to arguments which need simplification, written primitive!value, and symbolic applications $f$%$e$ which 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, leaving 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 $t_1 \rightarrow t_2$ 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

  LET $x$: int~$y+z$ IN $x+$mod $x$

  first evaluates $y+z$ and then evaluates $x+$mod $x$ with this value for $x$. The int may be omitted, thus

  LET $x$:~$y+z$ IN . . .

  The binding may be recursive, thus

  LET REC $f$: int→int ~ . . . IN . . .

  We allow $E$ WHERE $B$ as an abbreviation for LET $B$ IN $E$.
- function definitions: Functions are denoted by lambda expressions, for example

  λ $x$: int→int IN $x+$mod $x$

  which when applied to 3 evaluates 3+mod 3. If $T_1$ evaluates to $t_1$, $T_2$ evaluates to $t_2$, and $E$ is an expression of type $t_2$ provided that $N$ is a name of type $t_1$, then

  λ $N$: $T_1 \rightarrow T_2$ IN $E$

  is a function of type $t_1 \rightarrow t_2$. Functions of two or more arguments can be defined by using ×, for example

  λ $x$: int × $y$: bool →int IN . . .

  We allow the abbreviation $f$: ($i$: int→int) IS . . . for $f$: int→int ~ λ $i$: int→int IN . . .

An example may help to make this all more digestible:

```
LET REC fact : (n: int→int) IS
   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) IS
   λ n:int→int IN f(f n)
IN (twice fst) [[1, 2], 3]
```

which evaluates to fst(fst([[1, 2], 3], that is 1. We shall see later how we could define a polymorphic version of *twice* which would not be restricted to integer functions.

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 § 3.6.

*2.2 Bindings and declarations*

An unconventional feature of Pebble is that it treats bindings, such as $x\sim3$, 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 $x\sim3$. 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~$x+1$ evaluates to the binding $y\sim4$. The ~~expresssion~~ $x$: int~3 may be written more briefly $x$:~3. expression

The type of a binding is a declaration. Thus the binding expression $x$:~3 has as its type the declaration $x$: int. Bindings may be combined by pairing, just like any other values. Thus [$x$:~3, $b$:~true] is also a binding. After LET such a complex binding acts as two bindings "in parallel", binding both $x$ and $b$. Thus

LET $x$:~0 IN LET [$x$:~3, $y$:~$x$] 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 [$x$:~3, $b$:~true] is ($x$: int)×($b$: 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 $t_1 \times t_2$.

For convenience we have a syntactic sugar for combining bindings "in series". We write this $B_1$; $B_2$, which is short for [$B_1$, LET $B_1$ 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

λ $x$: int→int IN $x+1$

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

λ $x$: int × $y$: int → int IN $x+y$

which is of type int×int→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 $b$:~($x$:~3) IN LET $b$ IN $x$

which evaluates to 3. Another example is

```
LET f:~λ b: (x: int × y: int)→int IN
  LET b IN x+y IN
    f[x:~1, y:~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 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 to 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 § 3.1.

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

IMPORT *N* IN *E*

evaluates *E* in an environment in which *N* is the *only* name which is bound. For example:

LET *N*: ~ *B* IN IMPORT *N* IN LET *N* 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*. Without IMPORT, if *x* is missing from *B* we would pick up any *x* that happens to be in the current environment. IMPORT is so useful that we provide the syntactic sugar *B*$*x* for it.

### *2.3 Types*

We now explain how the kernel language 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*: *int*×*stack*→*stack*~. . ., etc.

- Multiple ADT – several predicates (zero or more) with a collection of associated operations.

  Example: *point* and *line* with particular operations:
  *intersection*: *line*×*line*→*point*~. . ., 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*×*line*→*point*, 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 gives the terminology for many sorted algebras.

| | Pascal | CLU | Mesa | Ada | Russell | ML | Algebra |
|---|---|---|---|---|---|---|---|
| Predicate | type | type | type | type | – | type | sort |
| Simple ADT | – | cluster | – | – | type | – | algebra |
| Multiple ADT | – | – | imple-mentation | package body | – | abstract type | algebra |
| ADT declaration | – | – | interface | package spec | – | – | signature |

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 ADT's, multiple ADT's 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.

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 maintains the traditional distinction between types and values, is being pursued by David MacQueen at Bell Labs, with some collaboration of one of us (RB). He has recently applied these ideas to the design of a module facility for ML [MacQueen 1984]. The theoretical basis for this work has been developed in [MacQueen and Sethi 1982, MacQueen, Plotkin and Sethi 1984].

### *2.4 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 have made this distinction in terms of program execution, lacking a mathematical theory. Recently Reynolds has offered a mathematical basis for this distinction [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 § 3.3 on generic types). But universal polymorphism may itself be handled in two ways: *explicit parameterisation* or *unification*.

> 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 [Reynolds 1974] and is used in Russell and CLU.
>
> Unification – 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 determine 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 expressions concerned [Milner 1978]; this method is used in ML [Gordon, Milner and Wadsworth]. For example we may write *reverse*(1, 2, 3, 4). Following Girard [Girard 1972] we may regard these type variables as universally quantified. The type of *reverse* would then be 'For all $t$: type . list($t$) → 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 in 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 unification 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. We hope to avoid the tedium in future versions of Pebble by sugar which automatically supplies a value for the type parameter.

For example, we might want to define a polymorphic function for reversing a pair, 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 a *dependent* type; we will explain this in the next section, and then we will be able to define the function *swap*.

### *2.5 Dependent types*

We now consider the idea of dependent type [Girard 1972, Demers and Donahue 1980]. 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 start with the former.

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

(type×type)→($t_1$×$t_2$→$t_2$×$t_1$)

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 the value of

($t_1$: type × $t_2$: type)→›($t_1$×$t_2$→$t_2$×$t_1$)

In order to have only one name-binding mechanism, we take this value to be

($t_1$: type × $t_2$: type)▸$c$

where $c$ is the closure which is the value of

λ$t_1$: type × $t_2$: type→type IN ($t_1$×$t_2$→$t_2$×$t_1$)

and ▸ 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)→›($t_1$×$t_2$→$t_2$×$t_1$) IS
  λ $x_1$:$t_1$ × $x_2$:$t_2$→$t_2$×$t_1$ IN [$x_2$, $x_1$]

Another example would be the list reversing function

REC *reverse*: ($t$:type)→›(*list t*→*list t*) IS
  λ *l*: *list t*→*list t* IN IF *l*= nil THEN *l* ELSE *append*[*reverse tail l*, [*head l*, nil]]

A similar ~~phenomemon~~ 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; thus [int, 3] and [bool, false] denote such pairs. The type of all such pairs may be written ($t$: type)××$t$. As we did with →›, we take its value to be ($t_1$: type × $t_2$: 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.

phenomenon

wrong font

A more realistic example might be

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

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

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

(ii) a transition function and an output function.

By "three types called *input*, *state* and *output*" we mean a binding of types to these names.

*2.6 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 $d_1 \bigstar f$ 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*~*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 [$d_1$~fst *e*, $d_2$~snd *e*] has type $d_1 \times d_2$;

otherwise, if the value of *D* is a dependent type of the form $d_1 \bigstar f$, then this must be reduced to the $d_1 \times d_2$ case by applying *f* to the binding ($d_1$~ fst *E*) to get $d_2$.

The type of a recursive binding REC *D*~*E* is just the value of *D*, provided that a somewhat complicated 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 $x$: int×int ~ [1 + 1, 0] IN fst $x$

has type int (and value 2). To show this, we first compute the type of the binding.

$x$: int×int ~ [1 + 1, 0] has type $x$: int×int because

$x$: int×int has type type and

$x$: int×int has value $x$: int×int and

[1 + 1,0] has type int×int

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

$x$: int×int ~ [1 + 1, 0] has value $x$~[2, 0]

We type-check fst $x$ in the new environment formed by adding [$x$: int×int] and [$x$~[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~int is such a name, and we need its binding to evaluate the rest of the expression.

Example:

LET $t$: type~int IN
  LET $x$: $t$~1 IN $x$+1

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

$t$: type~int has type $t$: type and value $t$~int

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

$x$: $t$~1 has type $x$: int and

$x$: $t$~1 has value $x$~1 and

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

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

λ *x*: int→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

λ *t*: type →› (*t*→*t*) IN λ *x*: *t*→*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

λ *x*: *t* → *t* IN *E*

in an environment enriched by [*t*: type] and [*t*~newconstant], where newconstant is a symbolic value of type type, distinct from all other symbolic values which we may invent. Under this regime a function such as

λ *t*: type →› (*t*→*t*) IN λ *x*: *t* → *t* IN *x*

will type-check (it has type denoted by *t*: type→›(*t*→*t*)) but

λ *t*: type →› (*t*→*t*) IN λ *x*: *t* → *t* IN *x*+1

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

Thus it is necessary that at type-checking time evaluation can give a symbolic result, since we may come across a newconstant. How do we apply one of the primitives to such a value? It will simply produce a value *w!e* which cannot be simplified. But what if the operator is symbolic? We introduce a special value constructing operator % to permit the application of a symbolic function to an argument. So 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 expressions.

## 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 Modula 2, 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 *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 don't wish to commit ourselves to the *RealFl* implementation, we can define a *parameterized* interface *MakeComplex*, which takes a *Real* parameter:

```
MakeComplex: (R: Real)→type IS
                     (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)) IS
                     [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)) IS
                     [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 don't need the flexibility of different kinds of complex numbers, we can dispense with the *Make* functions and simply write:

```
Cartesian: Complex~  [complex:~  R × R ;
                      mod:~      λ c: complex→R IN RealFl$sqrt((fst c)^2+(snd c)^2) , ...],
Polar: Complex~      [complex:~  R × R ;
                      mod:~      λ c: complex→R IN fst c , ...]
WHERE R:~RealFl$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) IS
                    (map:        (C1$complex × C2$complex → R$real) × ... ) ;
```

Note how 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:~       λ 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

```
intStackDecl: type ~    (stk: type ××
                         empty: stk ×
                         isEmpty: (stk→bool) ×
                         push: (int × stk→stk) ×
                         top: (stk→int) × ...)
```

is an abstract data type for a stack of ints. We have used a dependent ×× 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:

```
intStackOpsDecl: (stk: type →type) ~
                         (empty: stk ×
                          isEmpty: (stk→bool) ×
                          push: (int × stk→stk) ×
                          top: (stk→int) × ...)
```

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 ways 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:

```
StackOpsDecl: (elem: type × stk: type → type) IS
                         (empty: stk ×
                          isEmpty: (stk→bool) ×
                          push: (elem × stk→stk) ×
                          top: (stk→elem) × ...)
```

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

```
intStackOpsDecl: (stk: type →type) IS
                        StackOpsDecl(int, stk)
```

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

```
StackDecl: (elem: type→type) IS stk: type ××
                        StackOpsDecl[elem, stk]
```

and we can write the previous *intStackDecl* as

```
intStackDecl: type ~        StackDecl int
```

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

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

Here we have given the type of *list* but omitted the implementation, which is likely to be primitive.

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 is probably in this position, and will be written

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

This arrangement for the implementation leaves something to be desired in security. The client body is type-checked without any knowledge of the *list* implementation, and hence cannot compromise its security. However, the enclosing program which includes both looks like

```
LET StackFromList: ~--as above-- ..., Client: ~ --as above-- ... IN
                        ... Client(StackFromList) ...
```

and this program is in a position to construct a *list* int and pass it off as a *intStack*$*stk*. 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 × p: Password →›
                          AT: type ×× abs: (T→AT) × rep: (AT→T) ) ~ ... ;
```

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 only be constructed 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 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 § 2 and the formal one in § 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, 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 text of the implementation, but only to the interface. We want this to be true anyway.

Using *AbstractType* we can write a secure implementation:

```
StackFromList: (el: type)→>StackDecl el IS
                    LET (st:~ a.AT; abs:~a.abs; rep:~a.rep
                            WHERE a:~AbstractType(list el, 314159)) IN
                    (stk:~st;
                     procs:~    (empty:~abs nil;
                                 isEmpty: (s: stk→bool) IS 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 IS LET AbstractType(list el, 314159)) IN
                    (stk:~AT;
                     procs:~    (empty:~abs nil;
                                 isEmpty: (s: stk→bool) IS 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 XX
                        stk: type XX
                        StackOpsDecl[elem, stk]
```

This allows generic 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 a generic function is

```
Reverse: (S: aStackDecl×× x: S$stk→>S$stk) IS LET S IN
                    LET rev: (y: stk × z: stk→stk) IS
                                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*, *Print*~*string*$*Print*, "Hello" ]

A simple way to get this effect (using <> for string concatenation) is

```
AtomOps: t: type→type IS    Print: (t→list char)
atomT: type~                t: type ××
                            AtomOps(t)
atom: type~                 at: atomT ××
                            val: at$t

PrintAtom: (a: atom→list char) IS a$Print(a$val)
REC PrintList: (l: list atom→list char) IS
                    IF null l THEN "[]"
                    ELSE "["    <> PrintAtom (hd l)
                                <> ","
                                <> PrintList(tl l)
                                <> "]"
```

With this we can write

```
stringAtomT: ~       AtomT~[string, Print~PrintString] ;
hello:~              Atom~[stringAtomT, "Hello"]
intAtomT: ~          AtomT~[int, Print~PrintInt] ;
three:~              Atom~[numAtomT, 3]
```

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

This is fine for dealing with an individual value which can be turned into an *atom*, but suppose we want to print a list of ints. It isn't 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$t→list char) IS
                    LET a:~at$conv v IN a$Print(a$val)
REC PrintList: (at: atomX ×× l: list at$t→list char) IS
                    IF null l THEN "[]"
                    ELSE "["    <> PrintAtom[at, hd l]
                                <> ","
                                <> PrintList[at, tl l]
                                <> "]"
IntAsAtom: atomX~           ( t:~int,
                              conv: (v: t→atom) IS t:~int, Print:~PrintInt, val:~v )
```

*3.4 Union types*

There is a straightforward way to define a union type in Pebble:

$T_1 \oplus T_2 =$ *tag*: bool ×× *val*: (IF *tag* THEN $T_1$ ELSE $T_2$)

Unfortunately, it doesn't work, because the inference rules cannot evaluate the IF based on the rest of the program. For example, the expression

```
LET x: T1⊕T2~... IN
  IF x$tag=true THEN LET y: T1~x$val
```

will not type-check. There is no way to decompose a value of this type, and hence it is useless.

However, there are other ways to introduce unions. We have not completed a design, but here is a suggestive sketch. Following Cardelli [1984], we introduce a union or sum type parallel to the product type we already have. If $d_1$ and $d_2$ are declarations, then parallel to the labelled product $d_1 \times d_2$ there is a labelled **sum** $d_1 \oplus d_2$. An expression with type $d_1$ has type $d_1 \oplus d_2$, and so does an expression with type $d_2$. For example, if $D$:~($i$: int ⊕ $r$: real), then both $i$:~3 and $r$:~3.14 have type $D$.

Parallel to

LET *B* IN *E*

which decomposes a labelled product, is

CASE *E* OF *B*

which decomposes a labelled sum. If the type of $E$ is $n_1$: $t_1$ ⊕ $n_2$: $t_2$, the $B$ in CASE must have type

$n_1$: ($t_1 \rightarrow t$) ⊕ $n_2$: ($t_2 \rightarrow t$).

In other words, $B$ provides a suitable function for each case of $E$. The value of the CASE is obtained by chosing the proper function and applying it to rhs $E$. More precisely, in view of its type the value of $E$ must be $n_1$~$e_1$ or $n_2$~$e_2$. In the former case, the value of the CASE is $B\$n_1(e_1)$; in the latter it is $B\$n_2(e_2)$. To continue the previous example, after

```
LET f:~λ x: (i: int⊕ r: real) IN
  CASE x OF [
    i:~λ j: int→int IN j+1 ,
    r:~λ s: real→int IN fix(s) ]
```

$f$(*i*:~~int~~) has the value 4, and so does $f$(*r*:~4.1416).



*3.5 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 evaluation 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.

It is also interesting to note that union types are not necessary for meaningful recursive types, although they are very convenient. In fact, we can represent a list of integers as a function which takes no argument and returns an integer and another such function. In Pebble

```
LET REC IntList: type~(void→>int × IntList);
  REC empty: IntList~(λ IntList IN [error, empty]) IN
  ...
```

Now we can define a list of the first *n* integers:

```
LET REC f: int→IntList IS
  λ IntList IN IF i=0 THEN [0, empty] ELSE [i, f(i−1)]
```

The usefulness of this definition may well be questioned, but it does show that recursive types are a purely functional phenomenon.

### *3.6 Assignment*

Although Pebble as we have presented it is entirely applicative, it is easy to introduce imperative primitives. For example, we can 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 must be modified to carry around a store which := and ↑ can use to communicate.

In addition, steps must 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 much 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. § 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 1 gives a complete breakdown of the set of values.

| | | |
|---|---|---|
| $e$ | $e_0$ | viz true, false, 0, 1, 2, ..., etc. |
| | $f$ | `primitive`$(w)$ — where $w$ is +, ×, etc. |
| | | `closure`$(\rho, d, E)$ |
| | nil | |
| | $e, e$ | |
| | $b$ | $n \sim e$ |
| | | nil |
| | | $b, b$ |
| | | `fix`$(f, d)$ |
| | $t$ | $t_0$ viz bool, int, etc. |
| | | void |
| | | $t \times t$ |
| | | $t \rightarrow t$ |
| | | $d \blacktriangleright f$ |
| | | $d$ $n: t$ |
| | | void |
| | | $d \times d$ |
| | | $d \bigstar f$ |
| | $w!e$ | |
| | $f\%e$ | |

Table 1: Values

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

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

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 {nil}, $(w!e)$ means $\{(v_1!v_2) \mid v_1 \in w, v_2 \in e\}$ and so on for each value constructing operator.

The primitive constants of the value space are written in **this font**. Constructors such as `closure` are written in `this font`. Meta-variables which denote values or sets of values, possible of a given kind, are single lower-case 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 `primitive`(*w*), where *w* is some built-in function such as addition or multiplication of integers. They include functions on types such as ×. We write `primitive`(*w*) rather than just *w* to show that *w* is tagged as a primitive function. This is useful for matching purposes in our operational semantics; see § 5.

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

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

a *declaration* value, which gives the bound variables of the λ-expression;

a *body* expression, which is the expression (expressions are defined in § 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~1+2 IN ... will produce a binding *x*~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*~*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 declaration of the names being recursively defined and the function which represents one step of the recursive ~~defintition~~ definition (roughly, the functional whose fixed point is being computed). Details are given in § 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 $t_1 \times t_2$.

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

$d$, declarations. These are the type of bindings; for example, the type of $x$: int~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 $t \times t$).

$d \bigstar f$, a dependent version of $d \times t$. This is explained in § 2.5.

$d \blacktriangleright f$, a dependent version of $t \rightarrow t$. This is also explained in § 2.5.

$w!e$, the application of the primitive function $w$ to the value $e$. Such applications are values which may be simplified.

$f\%e$, the application of the symbolic value $f$ to the value $e$. This is explained in § 2.6.

We may define a relation ::: between values and types, analogous to the :: ('has type') relation between expressions and types defined in § 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 § 5. We first define a subsidiary relation 'is the type part of' between type values and declaration values; for example, int is the type part of $x$: int and int×bool is the type part of $x$: int×$p$: bool.

$t$ is the type part of $N$: $t$

void is the type part of void

$t_1 \times t_2$ is the type part of $d_1 \times d_2$ if $t_1$ is the type part of $d_1$ and $t_2$ is the type part of $t_2$.

Now for the definition of :::

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

  primitive(not) ::: bool→bool, and so on for other operators.

  primitive(×) ::: type×type→type,
  primitive(→) ::: type×type→type.

  closure($\rho$, $d$, $E$) ::: $t_1 \rightarrow t_2$ if $t_1$ is the type part of $d$ 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($d$, $f$) ::: $d$ if $f$ ::: $d \rightarrow d$.

- bool ::: type, int ::: type, type ::: type, void ::: type.

  $t_1 \times t_2$ ::: type if $t_1$ ::: type and $t_2$ ::: type.

- $N$: $t$ ::: type.

  $d \bigstar f$ ::: type if $f$ ::: $d \rightarrow$ type.

  $d \blacktriangleright f$ ::: type if $f$ ::: $d \rightarrow$ type.

- $w!e ::: t_2$ if $e ::: t_1$ and primitive$(w) ::: t_1 \rightarrow t_2$,

  $f!e ::: t_2$ if $e ::: t_1$ and $f ::: t_1 \rightarrow t_2$,
- $e_1, e_2 ::: d \bigstar f$ if supposing that $\vdash (f_{\#a \rightarrow type})(e_{1\#a}) \Rightarrow t_2$, then $e_1, e_2 ::: d \times t_2$.

  $f_1 ::: d \blacktriangleright f$ if for all $e$ such that $e ::: d$, if $\vdash (f_{\#a \rightarrow type})(e_{\#a}) \Rightarrow t_2$ then $(f_{1\#a \rightarrow type})(e_{\#a}) ::: t_2$.

The last clause might be written more simply by defining a notion of application for values, say $f\,e$, analogous to $F\,E$ for expressions, and writing

- $e_1, e_2 ::: d \bigstar f$ if $e_1, e_2 ::: d \times (f\,d)$.

  $f_1 ::: d \blacktriangleright f$ if for all $e$ such that $e ::: d$, $f_1\ e ::: f\,e$.

Now if $E \Rightarrow e$, we would like to have $e ::: t$ if and only if $E :: t$. But our type checking rules, which use symbolic evaluation, cannot always achieve this. A closure may have a certain type for all bindings, but symbolic evaluation may fail to show this. Consider for example

λ $x$: int →› (IF $x \leq x+1$ THEN int ELSE bool) IN $x$ :: int→int

This is not derivable from our typechecking rules because symbolic evaluation cannot show that $x \leq x+1$ for an arbitrary integer $x$. But the latter is true, so if $f$ is the value of the lambda expression we do get $f ::: \text{int} \rightarrow \text{int}$ 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*

curly ⊢

$E$ ::= bool | int | void | $E \times E$ | $E \rightarrow E$ | $E \rightarrow\!\!\geq E$ | $N{:}E$ | $E \times\!\times E$ | type | true | false | $I$ |
nil | $E, E$ | λ $E$ IN $E$ | $E{\sim}E$ | REC $E{\sim}E$ | $E$; $E$ | $N{:}{\sim}E$ | $N$ | IMPORT $N$ IN $E$ |
IF $E$ THEN $E$ ELSE $E$ | $E\ E$ | LET $E$ IN $E$ | typeOf $E$ | $(E)$ | $[E]$

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.

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 $T_1, T_2$; it is eliminated by expressions of the form fst $E$ or snd $E$.

The syntax presented in this way is shown in Table 2; a list of the notations used is given in Table 3. Table 4 shows some abbreviations which make Pebble more readable, for example eliminating the λ notation for function definitions.

| | *Type* | | *Introduction* | *Elimination* |
|---|---|---|---|---|
| T | bool | | true false | IF $E$ THEN $E_1$ ELSE $E_2$ |
| | int | | 0 1 2 ... | |
| | void | | nil | |
| | $T_1 \times T_2$ | | $E_1, E_2$ | fst $E$ snd $E$ |
| | $T \rightarrow T_0$ | F | $\lambda\ T$ IN $E$ | $F\ E$ |
| | $D \rightarrow\!\!> T_0$ | | primitives | |
| D | $N$: $T$ | B | $D \sim E$ | LET $B$ IN $E$ |
| | $D_1 \times D_2$ | | REC $D \sim E$ | |
| | $D_1 \times\!\times D_2$ | | $B_1, B_2$ | |
| | | | $B_1; B_2$ | |
| | | | $N$:~$E$ | |
| | type | | all types in the left column | typeOf $D$ |
| | | | $N$ | IMPORT $N$ IN $E$ |

Either round or square brackets may be used for grouping.
~~The "," ";" $\times$ and $\times\times$operators associate to the right, others to the left.~~
Precedence is: lowest IN, then "," ";", then ~, then $\rightarrow$ $\rightarrow\!\!>$, then $\times$ $\times\times$, highest :

All the operators associate to the right.

Table 2: Syntax

| *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 | $\lambda\ i$: int$\rightarrow$bool IN i>3 |

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

Table 3: Summary of abbreviations

| *Write* | *For* | *Example* |
|---|---|---|
| $N$ : $T$ IS $E$ | $N$ :~$\lambda\ T$ IN $E$ | $f$: ($i$: int $\times$ $x$: real)$\rightarrow$real IS $x^i$ |
| $[P_1, P_2]$ :~ $E$ | $P_1$ :~ fst $E$, $P_2$ :~ snd $E$ | $[i, j]$:~$QuotRem(7, 2)$ $\equiv$ $i$:~3, $j$:~1 |
| $B$ \$ $N$ | LET $B'$ :~ $B$ IN IMPORT $B'$ IN LET $B'$ IN $N$ | $[i$:~3, $x$:~$\pi]$\$$x$ $\equiv$ $\pi$ |
| $E$ WHERE $B$ | LET $B$ IN $E$ | $i$+4 WHERE $i$:~3 $\equiv$ 7 |

Table 4: Sugar

| *Type* | *Introduction* |
|---|---|
| bool<br>int | (a0) true :: bool⇒true (b0) false :: bool⇒false (c0) 0:: int⇒0 |
| void<br>$T_1 \times T_2$ | (a0) nil :: void⇒nil<br>(1) $E_1 :: t_1, E_2 :: t_2$ / (0) $[E_1, E_2] :: t_1 \times t_2 \Rightarrow [e_1, e_2]$ |
| $T_0 \rightarrow T$<br>$D \rightarrow\!\!> T$<br><br>*d = parameter decl*<br>$t_0$ *= parameter type*<br>$e_0$ *= argument value*<br>*t = result type*<br>$t_1$ *= type of* λ*-exp* | (1) { $T_1 \Rightarrow t_{11}, t_{11} \approx d \rightarrow t$, typeOf $d_{\#type} \Rightarrow t_0, t_0 \rightarrow t = t_1$<br>(2) **or** $T_1 \Rightarrow t_1$, $t_1 \approx d \blacktriangleright f$, $f_{\#d \rightarrow type}(newc_{\#d}) \Rightarrow t$ },<br>(3)<br>(4) LET $newc_{\#d}$ IN E :: t<br>*where* newc *is a new constant*<br>(0) $(\lambda\ T_1$ IN $E) :: t_1 \Rightarrow$ closure(ρ, d, E) |
| N: T<br>$D_1 \times D_2$<br>$D_1 \times\!\times D_2$ | (1) $D \Rightarrow d$, $d \approx$ void, E :: void, nil = b<br>(2) **or** $D \Rightarrow d$, $d \approx N : t$, E :: t, (N~e) = b<br>(3) **or** $D \Rightarrow d$, $d \approx d_1 \times d_2$, $[d_{1\#type} \sim$ fst E, $d_{2\#type} \sim$ snd E$] :: d \Rightarrow b$<br>(4) **or** $D \Rightarrow d_0$, $d_0 \approx d_1 \bigstar f$, $f_{\#d_1 \rightarrow type}(d_{1\#type} \sim$ fst E$) \Rightarrow d_2$,<br>(5) $d_1 \times d_2 = d$, $d_{\#type} \sim E :: d \Rightarrow b$<br>(0) $D \sim E :: d \Rightarrow b$<br><br>(a1) $D \Rightarrow d$, $(\lambda\ F' : D \rightarrow D$ IN LET F′ IN $d_{\#type} \sim E) \Rightarrow f$<br>(a2) $f_{\#d \rightarrow d}(f\ i\ x(f, d)_{\#d}) \Rightarrow b$<br>(a0) REC $D \sim E :: d \Rightarrow b$<br><br>(b1) $[B_1$, LET $B_1$ IN $B_2] :: t \Rightarrow b$ / (b0) $B_1 ; B_2 :: t \Rightarrow b$<br>(c1) $E :: t$ / (c0) $N :\sim E :: (N: t) \Rightarrow N \sim e$ |
| type | (a1) λ B′: D→type IN LET B′ IN T⇒*f*<br>(a0) $D \rightarrow\!\!> T ::$ type⇒d ▶ f<br>(b1) λ B′: D→type IN LET B′ IN T⇒*f*<br>(b0) $D \times\!\times T ::$ type⇒d★f<br>(c1) T :: type<br>(c0) $N : T ::$ type⇒N: t |
| *Names* | (1) $\rho(N) \approx t \sim e_0$<br>(2) $\{c_0 \rightsquigarrow e$ else $c_0 = e\}$<br>(0) $N :: t \Rightarrow c$ |

Table 5: Inference rules

| *Elimination* | *Type* |
|---|---|
| (1) E :: bool , $E_1$ :: $t$ , $E_2$ :: t , | bool |
| (2) {E⇒true , $E_1$⇒$e$ or E⇒false , $E_2$⇒$e$ } | int |
| (0) IF $E$ THEN $E_1$ ELSE $E_2$ :: t⇒e | |
| | void |
| (a0) fst :: $(t \times t_1) \rightarrow$t (b0) snd :: $(t_1 \times t) \rightarrow$t | $T_1 \times T_2$ |
| (1) { F :: $t_0 \rightarrow t$ , $E_0$ :: $t_0$ | $T_0 \rightarrow T$ |
| (2) or F :: $d \blacktriangleright f_1$ , $f_{1\#d \rightarrow type}(d\#type \sim E_0) \Rightarrow t$ }, | $D \rightarrow\!\!> T$ |
| (3) { f≈primitive($w$) , { w!$e_0$↝$e$ else w!e = $e$ } | *d = parameter decl* |
| (4) or f≈closure($\rho_0$,d,$E$),d#type~$E_0$⇒$b$, $\rho_0$⊢LET b#d IN E⇒$e$ | *$t_0$ = parameter type* |
| (5) else *in case f is a symbolic value* f%$e_0$ = $e$ } | *$e_0$ = argument value* |
| (0) $F\ E_0$ :: t⇒e | *t = result type* |
| | *$t_1$ = type of λ-exp* |
| (1) B:: void , E :: $t$⇒$e$ | N: T |
| (2) or B:: ($N$: $t_0$), rhs B ⇒$e_0$ , $\rho$[N:$t_0$~$e_0$] ⊢E :: $t$⇒$e$ | $D_1 \times D_2$ |
| (3) or B:: $d_1 \times d_2$, snd B ⇒$b_2$,LET fst B IN LET $b_2$#$d_1$ IN E :: $t$⇒$e$ | $D_1 \times\!\times D_2$ |
| (4) or B:: $d_1 \bigstar f$, $f_{\#d_1 \rightarrow type}$(fst B) ⇒$d_2$ , LET b#$d_1 \times d_2$ IN E :: $t$⇒$e$ | |
| (5) | |
| (0) LET $B$ IN $E$ :: t⇒e | |
| (b0) rhs :: ($N$: $t$)→t | |
| *these two are for convenience only* | |
| (1) { d≈void , void = $t$ | type |
| (2) or d≈$N$ : $t$ | |
| (3) or d≈$d_1 \times d_2$ , typeOf $d_1$#type × typeOf $d_2$#type ⇒$t$ | |
| (4) or d≈$d_1 \bigstar f$, typeOf!d = $t$}, | |
| (5) D :: type | |
| (0) typeOf D :: type⇒t | |
| | *Names* |
| (1) [N: $\rho$(N)] ⊢E :: $t$⇒$e$ | |
| (0) IMPORT $N$ IN $E$ :: t⇒e | |

Table 5: Inference rules (continued)

## Auxiliary rules

::
#

(1) $t \approx d_1 \star f$, $f_{\#d_1 \to type}(\text{fst } E) \Rightarrow t_2$, $E :: d_1 \times t_2$
(2) **or** $t \approx \text{typeOf!d}$, $d_{\#type} \sim E :: d$
(0) $E :: t$

(0) $e_{\#t} :: t \Rightarrow e$

↝
↝›

*for each ⟨arg, result⟩ pair in the primitive w*

(0) $w!e_0 \rightsquigarrow e$ (a0) $\text{fst!}[e, e_1] \rightsquigarrow e$ (b0) $\text{snd!}[e_1, e] \rightsquigarrow e$ (c0) $\text{rhs!}(N \sim e) \rightsquigarrow e$

(1) $e_0 \approx w!e_1$, $e_1 \rightsquigarrow e_2$, $w!e_2 \rightsquigarrow e$
(2) **or** $e_0 \approx \text{fix}(f, d)$, $f_{\#d \to d}(e_0 \#d) \Rightarrow e$
(0) $e_0 \rightsquigarrow e$

## Notation

$e_{\#t}$ is an *expression* with value e and type t.
$e_0 \rightsquigarrow e$ (*simplify*), $e_0 \rightsquigarrow e$ (*unroll*) define functions on *values*.
*Italics*: meta-variables bound by deterministic evaluator.
Value constants: type; value constructors: `closure`.

| | | |
|---|---|---|
| $N$ = name | | $i$ |
| $E$ = expression | $e$ = value | $gcd(i, 3)+1$ |
| $T$ = E with t value | $t$ = type value | int |
| $D$ = E with d value | $d$ = declaration value | $i$: int |
| $B$ = E with b value | $b$ = binding value | $i$: int~3 |
| $F$ = E with f value | $f$ = function value | $\lambda\, i$: int→bool IN $i$>3 |

Table 5: Inference rules (concluded)

## 5. Operational semantics

We have a precise operational semantics for Pebble, in the form of the set of inference rules in Table 5. 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 how each rule works, and finally show how to derive an efficient type-checker and evaluator from the rules.

### *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. We have in fact carried out this translation in part.

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 $assertion_1$, ..., $assertion_n$ and a conclusion $assertion_0$, written thus:

$$\frac{assertion_1, \ldots, assertion_n}{assertion_0}$$

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

$$\frac{assertion_{11}, \ldots, assertion_{1n_1} \textbf{ or } assertion_{21}, \ldots, assertion_{2n_2}}{assertion_0}$$

as an abbreviation for the two rules

$$\frac{assertion_{11}, \ldots, assertion_{1n_1}}{assertion_0} \qquad \frac{assertion_{21}, \ldots, assertion_{2n_2}}{assertion_0}$$

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 $\rho$, and is not written explicitly. If the environment for a premise is also $\rho$ (as it nearly always is), it is also omitted.

A *simple assertion* is one of:

1) $E :: t$ asserts that $E$ has 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*. For example, $e \approx t_1 \rightarrow t_2$; here $t_1 \rightarrow t_2$ is a format, with variables $t_1$ and $t_2$. If $e$ is int→bool, this assertion succeeds with $t_1 =$ int and $t_2 =$ bool.

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

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

5) $E :: format$ combines (1) 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 § 5.2.2.

8) $e_1 \rightsquigarrow\!> e_2$ asserts that $e_1$ *unrolls* to $e_2$, using the rule for unrolling `fix`. See § 5.2.5.

By convention we write a lower-case $e$ for the value of the expression $E$, and likewise for any other capital letter that stands for an expression. If a lower-case letter $x$ appears in an assertion but no premise is given to bind it, then the premise

$X \Rightarrow x$

is implied.

A reminder of our typographic conventions:

> We use capital letters for meta-variables denoting expressions, and lower-case letters for meta-variables denoting values; both may be subscripted. Thus expressions appear on the left of :: and ⇒ in assertions, and values everywhere else.
>
> Value constants are written **this way**: e.g., **true**, $x$: **int**.
>
> The value constructors that are not symbols are `primitive`, `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 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

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

as an abbreviation for

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

The fact that the rules are deterministic is important for another reason: they define a reasonably efficient deterministic program for evaluating expressions. We will have more to say about this in § 5.4.

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 one which is computed, along with its value, by all the other rules. We will say more about what this means for deterministic evaluation in § 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 ×I1, $t_1$ and $t_2$ are bound to the types of $E_1$ and $E_2$ respectively; they are used in ×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 ×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 that 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. In the text of the paper, we often drop the $\# t$ entirely, where no confusion is possible.

## *5.2 The rules*

The inference rules are organized like the syntax in Table 2, 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 $t_1 \rightarrow t_2$. 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 labeled by the same name, the less important ones are distinguished by letters a, b, ...; thus :Ia is the rule for REC. Auxiliary rules, with conclusions which are not part of the syntax, appear overleaf.

*5.2.1 Booleans, pairs and names*

The inference rules for booleans are extremely simple.

boolI

$$\frac{}{\text{(a0) true :: bool} \Rightarrow \text{true}} \qquad \frac{}{\text{(b0) false :: bool} \Rightarrow \text{false}}$$

boolE

$$\frac{\begin{array}{l}\text{(1) } E :: \text{bool}\ ,\ E_1 :: t\ ,\ E_2 :: t\ , \\ \text{(2) } \{\ E \Rightarrow \text{true}\ ,\ E_1 \Rightarrow e \text{ or } E \Rightarrow \text{false}\ ,\ E_2 \Rightarrow e\ \}\end{array}}{\text{(0) IF } E \text{ THEN } E_1 \text{ ELSE } E_2 :: t \Rightarrow e}$$

boolI tells us that the expressions true and false both have type bool and evaluate to true and false respectively; these rules have no premises, since the conclusions are always true. boolE says that the expression

IF $E$ THEN $E_1$ ELSE $E_2$

typechecks and has type $t$ if $E$ 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 $E$ is true, the value of $E_2$ if the value of $E$ is false. Thus

(A) IF true THEN 3 ELSE 5

has type int and value 3.

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 and inference rules (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 | boolIa |
| (A3) | 3 :: int⇒3 | intIc |
| (A4) | 5 :: int | intIc |

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.

×I

$$\frac{}{(a0)\ \text{nil} :: \text{void} \Rightarrow \text{nil}} \qquad \frac{(1)\ E_1 :: t_1 \ , E_2 :: t_2}{(0)\ [E_1, E_2] :: t_1 \times t_2 \Rightarrow [e_1, e_2]}$$

×E

$$\frac{}{(a0)\ \text{fst} :: (t \times t_1) \rightarrow t} \qquad \frac{}{(b0)\ \text{snd} :: (t_1 \times t) \rightarrow t}$$

×Ia says that nil has type void and value nil. ×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 $[e_1, e_2]$. ×E gives the (highly polymorphic) types of the primitives fst and snd that decompose pairs.

The rules for names are also straightforward, except for NI2, which is treated in § 5.2.5 since it is needed only for recursion.

NI

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

NE

$$\frac{(1)\ [N: \rho(N)] \vdash E :: t \Rightarrow e}{(0)\ \text{IMPORT } N \text{ IN } E :: t \Rightarrow e}$$

We can use NI to show

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

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

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

The IMPORT construct has a very simple rule, NE, which says that to evaluate IMPORT *N* IN *E*, evaluate *E* in an environment which contains *only* the current binding of *N*.

### *5.2.2 Functions*

The pivotal inference rules are →I (for defining a function by a λ-expression) and →E (for applying a function). The →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 →E) by evaluating the expression

(1) LET $d \sim E_0$ IN $E$

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:

→I

*d = parameter decl*
*$t_0$ = parameter type*
*$e_0$ = argument value*
*t = result type*
*$t_1$ = type of λ-exp*

$$\frac{(1)\ T_1 \Rightarrow t_{11} \ , t_{11} \approx d \rightarrow t \ , \text{typeOf d}_{\#\text{type}} \Rightarrow t_0 \ , t_0 \rightarrow t = t_1 \quad (4)\ \text{LET newc}_{\#d} \text{ IN E} :: t \quad \textit{where newc is a new constant}}{(0)\ \qquad (\lambda\ T_1 \text{ IN } E) :: t_1 \Rightarrow \text{closure}(\rho, d, E)}$$

The notes on the left of explain the meaning of the meta-variables. The expression $T_1$ in the λ roughly gives the type of the entire λ-expression. Thus

(B) λ *i*: 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 $t_{11}$; 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 $d \rightarrow i$, and computes first $t_0$ as typeOf *d* (using typeE), and then $t_1$ as $t_0 \rightarrow t$.

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

(2) LET newc$_{*d}$ IN *E*

has the result type *t*, where newc is a new constant, about which we know nothing except that its type is *d*. In other words, newc is a binding for the names in *d*, in ~~whch~~ which each name has the type assigned to it by *d*. For our example (B), we have

(3) LET newc1$_{*i\ \mathrm{int}}$ IN *i*+1

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

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

Using this, (3) has type int if

$\rho$[*i*: int~rhs!newc1]⊢*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 →E1), provided that [*i*, 1] has the argument type of plus. Since

plus :: int×int→int

we have the desired result if [*i*, 1] :: int×int. Using ×I this is true if *i* :: int and 1 :: int. The latter is immediate, since 1 is a primitive. According to NE, the former is true if $\rho(i) \approx \mathrm{int}{\sim}e_0$. But in fact $\rho(i)$=int~rhs!newc1, so we are done.

We can write this argument more formally as follows:

| | | |
|---|---|---|
| (B1) | $\rho \vdash$ LET newc1$_{*i\ \mathrm{int}}$ IN *i*+1 :: int | 2, :E |
| (B2) | $\rho_1 \vdash i+1$ :: int<br>where $\rho_1 = \rho$[*i*: int~rhs!newc1] | 3, →E |
| (B3) | $\rho_1 \vdash$ plus :: *t*→int, [*i*, 1] :: *t* | 4, 5 |
| (B4) | $\rho_1 \vdash$ plus :: int×int→int | primitive |
| (B5) | $\rho_1 \vdash$ [*i*, 1]:: int×int | 5, ×E |
| (B6) | $\rho_1 \vdash i$ :: int, 1 :: int | 7, NE, primitive |
| (B7) | $\rho_1(i) \approx \mathrm{int}{\sim}e_0$ | inspection |

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

→E
$$\frac{\begin{array}{l}(1)\ F :: t_0 \to t\ ,\ E_0 :: t_0\ , \\ (3)\{\ f \approx \mathtt{primitive}(w)\ , \qquad \{\ w!e_0 \rightsquigarrow e \text{ else } w!e = e\ \} \\ (4)\text{or } f \approx \mathtt{closure}(\rho_0, d, E),\ d_{\#\mathrm{type}} \sim E_0 \Rightarrow b,\ \rho_0 \vdash \text{LET } b_{\#d} \text{ IN } E \Rightarrow e \\ (5)\text{else } \textit{in case } f \textit{ is a symbolic value} \qquad f\%e_0 = e\ \}\end{array}}{(0)\qquad F\,E_0 :: t \Rightarrow e}$$

The type-checking is done by →E1, which simply checks that the argument $E_0$ has the parameter type $t_0$ of the function. There are three cases for evaluation, depending on whether $f$ is a primitive, a closure, or a symbolic value.

If $f$ is primitive($w$), →E3 tries to use the ⇝ rules for evaluating primitives to obtain the value of the primitive when applied to the argument value $e_0$.

⇝
$$\frac{\textit{for each } \langle \textit{arg, result} \rangle \textit{ pair in the primitive } w}{(0)\qquad w!e_0 \rightsquigarrow e}$$

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. If no ⇝ rule is applicable, the value is just $w!e_0$, i.e., a more complex symbolic value.

Thus the ⇝ 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 ⇝ 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 ⇝a-c illustrate.

⇝a-c
$$(a0)\ \mathsf{fst}![e, e_1] \rightsquigarrow e \qquad (b0)\ \mathsf{snd}![e_1, e] \rightsquigarrow e \qquad (c0)\ \mathsf{rhs}!(N \sim e) \rightsquigarrow e$$

Note that the soundness of the type system depends on consistency between the types of a primitive (as expressed in rules like ×Ea-b), and the ⇝ rules for that primitive (⇝a-b for fst and snd). For each primitive, a proof is required that the ⇝ rules give a result for every argument of the proper type, and that the result is of the proper type.

If $f$ is closure($\rho_0$, $d$, $E$), →E4 first computes a binding $b = d_{\#\mathrm{type}} \sim E_0$ from the argument $E_0$ in the current environment, and then evaluates the closure body $E$ in the closure environment $\rho_0$ augmented by $b$. Note the parallel with →I4, which is identical except that the unknown argument binding $\mathsf{newc}_{\#d}$ replaces the actual argument binding $d_{\#\mathrm{type}} \sim E_0$. The success of the type-check made by →I4 when $f$ was constructed ensures that the LET in →E4 will type-check.

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 it is left 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.

$$\rightarrow\mathbf{I} \qquad \begin{array}{l} (2)\ T_1 \Rightarrow t_1\ ,\ t_1 \approx d \blacktriangleright f,\ f_{\#d\rightarrow type}(newc_{\#d}) \Rightarrow t\ , \\ (4)\ \text{LET } newc_{\#d} \text{ IN } E :: t \quad \textit{where } newc \textit{ is a new constant} \end{array}$$

$$\overline{(0) \qquad (\lambda\ T_1 \text{ IN } E) :: t_1 \Rightarrow closure(\rho, d, E)}$$

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) $swap: \sim\lambda\ (t_1{:}type \times t_2{:}type) \rightarrow\!> (t_1 \times t_2 \rightarrow t_2 \times t_1)$ 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 *swap* (following the first λ) evaluates (by typeIa) to

(4) $(t_1{:}type \times t_2{:}type) \blacktriangleright$
$closure(\rho, B'{:}\ (t_1{:}type \times t_2{:}type), \text{LET } B' \text{ IN } t_1 \times t_2 \rightarrow t_2 \times t_1)$

In this case the parameter type of *swap* is just $(t_1{:}type \times t_2{:}type)$; 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 function 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 the same type.

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 newc1, a new constant which must be the same here and in the instantiation of →I4. In this example, this application yields

rhs!fst!newc1×rhs!snd!newc1→rhs!snd!newc1×rhs!fst!newc1

which we call *t*.

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

(C1) $\rho \vdash$ LET $newc1_{\#t_1:type \times t_2:type}$ IN — 2, :E
$\lambda\ x_1{:}t_1 \times x_2{:}t_2 \rightarrow t_2 \times t_1$ IN $[x_2, x_1]$ ::
rhs!fst!newc1×rhs!snd!newc1→rhs!snd!newc1×rhs!fst!newc1

(C2) $\rho_1 \vdash \lambda\ x_1{:}t_1 \times x_2{:}t_2 \rightarrow t_2 \times t_1$ IN $[x_2, x_1]$ :: — equality, 3, →I
rhs!fst!newc1×rhs!snd!newc1→rhs!snd!newc1×rhs!fst!newc1
where $\rho_1 = \rho[t_1{:}type \sim rhs!fst!newc1\ \ t_2{:}type \sim rhs!snd!newc1]$

(C3) $\rho_1 \vdash$ LET $newc2_{\#x_1:rhs!fst!newc1 \times x_2:rhs!snd!newc1}$ IN $[x_2, x_1]$ :: — 4, :E
rhs!snd!newc1×rhs!fst!newc1

(C4) $\rho_2 \vdash [x_2, x_1]$ :: rhs!snd!newc1×rhs!fst!newc1 — 5, ×E
where $\rho_2 = \rho_1[x_1{:}rhs!fst!newc1 \sim rhs!fst!newc2,$
$x_2{:}rhs!snd!newc1 \sim rhs!snd!newc2]$

(C5) $\rho_2 \vdash x_2 ::$ rhs!snd!newc1, $\rho_2 \vdash x_1 ::$ rhs!fst!newc1 6, NE

(C6) $\rho_2(x_2) \approx$ rhs!snd!newc1 $\sim e_{02}$, $\rho_2(x_1) \approx$ rhs!fst!newc1 $\sim e_{01}$ inspection

Observe that we carry symbolic forms (e.g. rhs!snd!newc1) of the values of the arguments for functions whose bodies are being typechecked. 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 and the fact that we generate a *different* newc constant for each λ-expression.

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

→E
(2) $F :: d \blacktriangleright f_1$, $f_{1\#d\to type}(d_{\#type} \sim E_0) \Rightarrow t$,
(3) { $f \approx primitive(w)$, { $w!e_0 \hookrightarrow e$ **else** $w!e = e$ }
(4) **or** $f \approx closure(\rho_0, d, E), d_{\#type} \sim E_0 \Rightarrow b$, $\rho_0 \vdash$ LET $b_{\#d}$ IN $E \Rightarrow e$
(5) **else** *in case f is a symbolic value* $f\%e_0 = e$ }
(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 $f$ to the argument binding $d_{\#type} \sim E_0$. 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 $f$ to $d_{\#type} \sim E_0$ rather than to $E_0$ because typeIa, which constructs $d \blacktriangleright f$, expects a binding as the argument of $f$. The reason for this is that in →E2 we don't 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 $d_{\#type} \sim E_0$ that checks the type of the argument; there is no need for the explicit check $E_0 :: t$ of →E1.

### *5.2.4 Bindings and declarations*

The main rules for bindings show how to typecheck and evaluate a binding made from a declaration and an expression (:I) and how to use a binding in a LET to modify the environment in which a subexpression is evaluated (:E). The tricky case of recursive bindings (:Ia and NI2) is discussed in the next section. Rules :Ib and :Ic define the ";" and :~ abbreviations; both are very simple.

The rule for $D \sim E$ has four cases, depending on the form of the declaration value.

:I
(1) $D \Rightarrow d$, $d \approx$ void, $E ::$ void, nil $= b$
(2) **or** $D \Rightarrow d$, $d \approx N : t$, $E :: t$, $(N \sim e) = b$
(3) **or** $D \Rightarrow d$, $d \approx d_1 \times d_2$, $[d_{1\#type} \sim$ fst E, $d_{2\#type} \sim$ snd E] $:: d \Rightarrow b$
(4) **or** $D \Rightarrow d_0$, $d_0 \approx d_1 \bigstar f$, $f_{\#d_1 \to type}(d_{1\#type} \sim$ fst E) $\Rightarrow d_2$,
(5) $d_1 \times d_2 = d$, $d_{\#type} \sim E :: d \Rightarrow b$
(0) $D \sim E :: d \Rightarrow b$

If the declaration is void, $E$ must have type void also, and the result is nil. If it is $N$: $t$, $E$ must have type $t$, and the result is the binding value $N \sim e$. These are the base cases. If the declaration is $d_1 \times d_2$, $E$ must have a × type, 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*~3, *x*~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, $d_0 = d_1 \bigstar f$ can be converted to $d_1 \times d_2$ by applying $f$ 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*~int, *x*~3]. In this case the type of the binding is not $d_0$, but the simpler cross type $d = d_1 \times d_2$.

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

**:E**

(1) B:: **void** , E :: $t \Rightarrow e$

(2) **or** B:: (*N*: $t_0$), rhs B $\Rightarrow e_0$ , $\rho$[N:$t_0$~$e_0$] ⊢ E :: $t \Rightarrow e$

(3) **or** B:: $d_1 \times d_2$, snd B $\Rightarrow b_2$, LET fst B IN LET $b_{2\#d_2}$ IN E :: $t \Rightarrow e$

(4) **or** B:: $d_1 \bigstar f$, $f_{\#d_1 \to type}$(fst B) $\Rightarrow d_2$ , LET $b_{\#d_1 \times d_2}$ IN E :: $t \Rightarrow e$

---

(0) LET *B* IN *E* :: t⇒e

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

λ *t*: type ×× *x*: *t* → int IN *E*

→I4 requires a proof of

LET $newc_{\#d_1 \bigstar f}$ IN *E* :: int

where $d_1 \bigstar f$ is the value of *t*: type ×× *x*: *t*. :E4 reduces this to

LET $newc_{\#t: type \times x: fst\ newc}$ IN *E* :: int

*5.2.5 Recursion*

Recursion is handled by a fixed point constructor in the value space, fix($f$, $d$). If $f$ is a function with type $d \to d$, then fix($f$, $d$) has type $d$ and is the fixed point of $f$, i.e.,

$f_{\#d \to d}(\mathrm{fix}(f, d)_{\#d}) \Rightarrow \mathrm{fix}(f, d)$.

The novelty is in the treatment of mutual recursion: $d$ may declare any number of names, and correspondingly fix($f$, $d$) binds all these names. A fix($f$, $d$) value is the result of evaluating a REC $D \sim E$ binding. For example,

REC $g$: (int→int) × $h$: (int→int)~[
  λ $x$: int→int IN IF $x=0$ THEN 1 ELSE $x$*$h(x/2)$,
  λ $y$: int→int IN IF $y<2$ THEN 0 ELSE $g(y-2)$ ]

has type

$g$: (int→int) × $h$: (int→int).

Its value is a binding for $g$ and $h$ in which their values are the closures we would expect, with an environment $\rho_{gh}$ 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:

[ $g$~closure($\rho$ , $x$: int, IF $x=0$ THEN 1 ELSE $x$*$h(x/2)$),
  $h$~closure($\rho$ , $y$: int, IF $y<2$ THEN 0 ELSE $g(y-2)$) ]

where

$\rho = \rho[g$: int→int~rhs!fst!fix($f$, $d$), $h$: int→int~rsh!snd!fix($f$, $d$)],

where

$d = g$: (int→int) × $h$: (int→int)

and

$f$=closure($\rho$, $F'$: $d$, LET $F'$ IN $d$~[
  λ $x$: int→int IN IF $x=0$ THEN 1 ELSE $x$*$h(x/2)$,
  λ $y$: int→int IN IF $y<2$ THEN 0 ELSE $g(y-2)$ ] )

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

$$\rightsquigarrow> \qquad \frac{\begin{array}{ll}(1) & e_0 \approx w!e_1\ ,\ e_1 \rightsquigarrow> e_2\ ,\ w!e_2 \rightsquigarrow e \\ (2) & \text{or } e_0 \approx \mathrm{fix}(f, d)\ ,\ f_{\#d \to d}(e_{0\#d}) \Rightarrow e\end{array}}{(0) \qquad e_0 \rightsquigarrow> e}$$

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

fix($f$, $d$)↝>[$g$~closure($\rho$ , $x$: int, ...), $h$~closure($\rho$ , $y$: int, ...)]

using ↝>2 and →E, and then simplifying rhs!fst!fix($f$, $d$) to closure($\rho_{gh}$, $x$: int, ...) using ↝>1, ↝a and ↝b. Thus, each time $g$ or $h$ is looked up in $\rho_{gh}$, the NI and ↝> 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→int~λ *N*: int→int IN
IF *n*<2 THEN *m* ELSE *P*(*N*−2)

We can write this more compactly as

REC DP~L

where

DP=P: int→int,
L=λ *n*: int→int IN EXP,
EXP=(IF *n*<2 THEN *n* ELSE *P*(*n*−2))

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

P: int→int~closure($\rho_{fp}$, *n*: int, 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 fix(*f*, *dp*) 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 $\rho_{fp}$, which contains the necessary fix value for *P* (D7-10). Now evaluating the λ to obtain a closure value for *P* that contains $\rho_{fp}$ is easy (D12-13).

| | | |
|---|---|---|
| (D1) | $\rho\vdash$REC DP~L :: dp⇒bp | :Ia, 1a, 3, 5 |
| (D1a) | $\rho\vdash$DP⇒dp | typeIc, 2 |
| (D2) | $\rho\vdash$P: int→int=DP | definition |
| (D3) | $\rho\vdash$(λ *F*′: DP→DP IN LET *F*′ IN dp~L)⇒f | →I, 4 |
| (D4) | closure($\rho$, *F*′: dp, LET *F*′ IN dp~L)=f | definition |
| (D5) | $\rho\vdash$f(fix(f, dp))⇒bp | →E, 6 |
| (D6) | $\rho\vdash$*F*′: dp~fix(f, dp)⇒bf,<br>$\rho\vdash$LET bf IN LET *F*′ IN dp~L⇒bp | #, :I, 7, :E,8 |
| (D7) | *F*′: dp~fix(f, dp)=bf | definition |
| (D8) | $\rho_f\vdash$LET *F*′ IN dp~L⇒bp<br>where $\rho_f=\rho$[*F*′: dp~fix(f, dp)] | :E, 9, 10 |
| (D9) | $\rho_f\vdash$rhs *F*′⇒rhs!fix(f, dp) | →E, NI |
| (D10) | $\rho_{fp}\vdash$dp~L⇒bp<br>where $\rho_{fp}=\rho_f$[*P*: int→int~rhs!fix(f, dp)] | :I, 11, 12 |
| (D11) | $\rho_{fp}\vdash$L⇒closure($\rho_{fp}$, *n*: int, EXP) | →I |
| (D12) | *P*~closure($\rho_{fp}$, *n*: int, 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 × *j*: int~[*j*+1, 0],

which binds *i*:~1 and *j*:~0. However, it may not terminate. For instance, consider

REC *i*: int × *j*: int~[*j*+1, *i*]

Now we look at an application of *P*:

(E) LET ( REC *P*: int→int~λ *N*: int→int IN
IF *n*<2 THEN *m* 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* (E1-2). The application becomes a LET after *P* and 3 are evaluated (E3-5). This results in an environment $\rho_{n3}$ in which *n*~3, so we need to evaluate *P*(*n*−2) (E6-7). Looking up *P* we find a value which can be unrolled (E8-9) to obtain the recursive value closure($\rho_{fp}$, *n*: int, EXP) again (E10-11). Since *n*−2⇒1 (E12), we get the answer without any more recursion (E13-15).

| | | |
|---|---|---|
| (E1) | $\rho$⊢LET REC DP~L IN *P*(3)) :: int⇒1 | :E, D, 2 |
| (E2) | $\rho_p$⊢*P*(3) :: int⇒1<br>where $\rho_p = \rho$[*P*: int→int~closure($\rho_{fp}$, *n*: int, EXP)] | →E, 3, 4, 5 |
| (E3) | $\rho_p$⊢*P*::int→int⇒closure($\rho_{fp}$, *n*: int, EXP) | NI |
| (E4) | $\rho_p$⊢*n*: int~3⇒*n*~3 | :I, intI |
| (E5) | $\rho_{fp}$⊢LET *n*~3 IN EXP:: int⇒1 | :E, 6 |
| (E6) | $\rho_{n3}$⊢IF *n*<2 THEN *n* ELSE *P*(*n*−2):: int⇒1<br>where $\rho_{n3} = \rho_{fp}$[*n*: int~3] | boolE, 7 |
| (E7) | $\rho_{n3}$⊢*P*(*n*−2):: int⇒1 | →E, 8, 12, 13 |
| (E8) | $\rho_{n3}$⊢*P*:: int→int⇒closure($\rho_{fp}$, *n*: int, EXP) | NI, 9 |
| (E9) | rhs!fix(f, dp)↝>closure($\rho_{fp}$, *n*: int, EXP) | ↝>, 11 |
| (E11) | rhs!bp↝closure($\rho_{fp}$, *n*: int, EXP) | ↝b |
| (E12) | $\rho_{n3}$⊢*n*−2:: int⇒1 | NI, →E, ↝ |
| (E13) | $\rho_{fp}$⊢LET *n*~1 IN EXP:: int⇒1 | :E, 14 |
| (E14) | $\rho_{n1}$⊢IF *n*<2 THEN *n* ELSE *P*(*n*−2):: int⇒1<br>where $\rho_{n1} = \rho_{fp}$[*n*: int~1] | boolE, 15 |
| (E15) | $\rho_{n1}$⊢*n*:: int⇒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*★*f* and typeOf!(*d*★*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.

::
(1) $t \approx d_1 \bigstar f,\ f_{\#d_1 \to type}\,(\text{fst } E) \Rightarrow t_2,\ E :: d_1 \times t_2$
(2) or $t \approx \text{typeOf!}d,\ d_{\#type} \sim E :: d$
(0) $E :: t$

| *Introduction* | *Elimination* |
|---|---|
| (a0) true :: bool | (1) E :: bool , $E_1$ :: $t$ , $E_2$ :: t , |
| (b0) false :: bool | (2) |
| (c0) 0:: int | (0) IF $E$ THEN $E_1$ ELSE $E_2$ :: t |
| (a0) nil :: void | (a0) fst :: $(t \times t_1) \rightarrow$t |
| (1) $E_1$ :: $t_1$ , $E_2$ :: $t_2$ | (b0) snd :: $(t_1 \times t) \rightarrow$t |
| (0) $[E_1, E_2]$ :: $t_1 \times t_2$ | |
| (1) { $T_1 \Rightarrow t_{11}$ , $t_{11} \approx d \rightarrow t$ , typeOf $d_{\#\text{type}} \Rightarrow t_0$ , $t_0 \rightarrow t = t_1$ | (1) { F :: $t_0 \rightarrow t$ , $E_0$ :: $t_0$ |
| (2) **or** $T_1 \Rightarrow t_1$ , $t_1 \approx d \blacktriangleright f$, $f_{\#d \rightarrow \text{type}}(\text{newc}_{\#d}) \Rightarrow t$ } , | (2) **or** F :: $d \blacktriangleright f_1$ , $f_{1\#d \rightarrow \text{type}}(d_{\#\text{type}} \sim E_0) \Rightarrow t$ } , |
| (3) | (3) |
| (4) LET $\text{newc}_{\#d}$ IN E :: t<br>*where* newc *is a new constant* | (4)<br>(5) |
| (0) ($\lambda$ $T_1$ IN $E$) :: $t_1$ | (0) $F\ E_0$ :: t$\Rightarrow$e |
| (1) D$\Rightarrow d$, d$\approx$void , E :: void , | (1) B:: void , E :: $t$ |
| (2) **or** D$\Rightarrow d$, d$\approx N : t$, E :: t , | (2) **or** B:: (N: $t_0$), rhs B $\Rightarrow e_0$ , $\rho$[N:$t_0 \sim e_0$] $\vdash$E :: $t$ |
| (3) **or** D$\Rightarrow d$, d$\approx d_1 \times d_2$, [$d_{1\#\text{type}} \sim$fst E,$d_{2\#\text{type}} \sim$snd E]::d | (3) **or** B:: $d_1 \times d_2$, snd B $\Rightarrow b_2$,LET fst B IN LET $b_{2\#d_2}$ IN E :: $t$ |
| (4) **or** D$\Rightarrow d_0$, $d_0 \approx d_1 \bigstar f$, $f_{\#d_1 \rightarrow \text{type}}(d_{1\#\text{type}} \sim \text{fst E}) \Rightarrow d_2$ , | (4) **or** B:: $d_1 \bigstar f$, $f_{\#d_1 \rightarrow \text{type}}(\text{fst B}) \Rightarrow d_2$ , LET $b_{\#d_1 \times d_2}$ IN E :: $t$ |
| (5) $d_1 \times d_2 = d$, $d_{\#\text{type}} \sim$E::d | (5) |
| (0) $D \sim E$ :: d$\Rightarrow$b | (0) LET $B$ IN $E$ :: t$\Rightarrow$e |
| (a1) D $\Rightarrow d$<br>(a2)<br>(a0) REC $D \sim E$ :: d | (b0) rhs :: $(N: t) \rightarrow$t |
| (b1) [$B_1$ , LET $B_1$ IN $B_2$] :: $t$<br>(b0) $B_1$ ; $B_2$ :: t | (c1) E :: $t$<br>(c0) $N :\sim E$ :: (N: t) |
| | *these two are for convenience only* |
| (a1) $\lambda$ B': D$\rightarrow$type IN LET B' IN T<br>(a0) $D \rightarrow\!\!> T$ :: type | (1)<br>(2)<br>(3) |
| (b1) $\lambda$ B': D$\rightarrow$type IN LET B' IN T<br>(b0) $D \times\!\!\times T$ :: type | (4)<br>(5) D :: type |
| (c1) T :: type<br>(c0) $N : T$ :: type | (0) typeOf D :: type |
| (1) $\rho$(N)$\approx t \sim e_0$<br>(2)<br>(0) $N$ :: t | (1) [N: $\rho$(N)] $\vdash$E :: $t$<br>(0) IMPORT $N$ IN $E$ :: t |

Table 6a: Inference rules for type checking only

*Introduction* | *Elimination*

(a0) true ⇒true (b0) false ⇒false (c0) 0 ⇒0

(a0) nil ⇒nil (0) $[E_1, E_2]$ ⇒$[e_1, e_2]$ (1) $E_1 :: t_1$ , $E_2 :: t_2$

(1) { $T_1 \Rightarrow t_{11}$ , $t_{11} \approx d \rightarrow t$
(2) or $T_1 \Rightarrow t_1$ , $t_1 \approx d \blacktriangleright f$ }
(3)
(4) LET newc#d IN E :: t
*where* newc *is a new constant*
(0) $(\lambda\ T_1$ IN $E)$ ⇒closure(ρ, d, E)

(1) D⇒$d$, d≈void , nil = $b$
(2) or D⇒$d$, d≈$N : t$ , (N~e) = $b$
(3) or D⇒$d$, d≈$d_1 \times d_2$, [$d_1$#type~fst E,$d_2$#type~snd E]::d⇒$b$
(4) or D⇒$d_0$, $d_0 \approx d_1 \bigstar f$, f#$d_1$→type($d_1$#type~fst E ) ⇒$d_2$ ,
(5) $d_1 \times d_2 = d$, d#type~E::d⇒$b$
(0) $D \sim E$ ⇒b

(a1) D ⇒$d$, (λ F′:D→D IN LET F′ IN d#type~E )⇒$f$
(a2) f#d→d( f i x(f, d)#d)⇒b
(a0) REC $D \sim E$ ⇒b

(b1) [$B_1$ , LET $B_1$ IN $B_2$] ⇒$b$
(b0) $B_1$ ; $B_2$ ⇒b
(c1) E :: $t$
(c0) $N :\sim E$ :: (N: t)⇒N~e

(a1) λ B′: D→type IN LET B′ IN T⇒$f$
(a0) $D \rightarrow\!\!\succ T$ ⇒d▶f
(b1) λ B′: D→type IN LET B′ IN T⇒$f$
(b0) $D \times\!\!\times T$ ⇒d★f
(c1)
(c0) $N : T$ ⇒N: t

(1) ρ(N)≈$t$~$e_0$
(2) {$e_0 \rightsquigarrow e$ else $e_0 = e$}
(0) $N$ ⇒e

(1)
(2) {E⇒true , $E_1$⇒$e$ or E⇒false , $E_2$⇒$e$ }
(0) IF $E$ THEN $E_1$ ELSE $E_2$ ⇒e

(a0) (b0)

(1)
(2)
(3) { f≈primitive($w$) , { w!$e_0 \rightsquigarrow e$ else w!e = $e$ }
(4) or f≈closure($\rho_0$,d,$E$),d#type~$E_0$⇒$b$, $\rho_0$⊢LET b#d IN E ⇒$e$
(5) else *in case f is a symbolic value* f%$e_0$ = $e$ }
(0) $F\ E_0$ ⇒e

(1) B:: void , E ⇒$e$
(2) or B:: ($N$: $t_0$), rhs B ⇒$e_0$ , ρ[N:$t_0$~$e_0$] ⊢E ⇒$e$
(3) or B:: $d_1 \times d_2$, snd B ⇒$b_2$,LET fst B IN LET $b_2$#$d_2$ IN E ⇒$e$
(4) or B:: $d_1 \bigstar f$, f#$d_1$→type(fst B) ⇒$d_2$ , LET b#$d_1 \times d_2$ IN E ⇒$e$
(5)
(0) LET $B$ IN $E$ ⇒e

(b0)

*these two are for convenience only*

(1) { d≈void , void = $t$
(2) or d≈$N$ : $t$
(3) or d≈$d_1 \times d_2$, typeOf $d_1$#type × typeOf $d_2$#type ⇒$t$
(4) or d≈$d_1 \bigstar f$, typeOf!d = $t$} ,
(5)
(0) typeOf D ⇒t

(1) [N: ρ(N)] ⊢E ⇒$e$
(0) IMPORT $N$ IN $E$ ⇒e

Table 6b: Inference rules for evaluation only

```
type VR = record case tag of
                          boolean:        (v: (true, false) ),
                          typeConst:      (v: (bool, int, void, type)),
                          primitive:      (w: Primitive),
                          nil:            ( ),
{(e1 e2, )}               pair:           (e1, e2: V),
                          closure:        (rho: Binding, domain: Decl, body: Ex),
{n:~e}                    binding:        (n: Name, e: V),
{w!e}                     bang:           (w:V {Primitive}, e: V),
{t1×t2}                   cross:          (t1, t2: Type),
{d★f}                     dcross:         (d: Decl, f: V {closure}),
{t→t0}                    arrow:          (domain, range: Type),
{d▶f}                     darrow:         (domain: Decl, frange: V {closure}),
{n: t}                    sdecl:          (n: Name, t: Type),
{f%e}                     symbolicApply:  (f, e: V),
                          end

type V = pointer to VR;

type Binding = VR         {either         V {binding}
                           or             V {pair[Binding, Binding]} }

type Decl = VR            { either        V {sdecl}
                           or             V {nil}
                           or             V {cross[Decl, Decl]}
                           or             V {dcross[Decl, V.closure]} }

type Type = VR            { either        V {typeConst}
                           or             V {cross}
                           or             V {dcross}
                           or             V {arrow}
                           or             V {darrow}
                           or             V {decl} }

{or any of these could be V {bang} or V {symbolicApply} }

type ExR = record case tag of
                          constant:       (c: (true, false, ...)),
{IF E THEN E1 ELSE E2}    if:             (E: Ex, E1, E2: Ex),
{E1, E2}                  pair:           (E1, E2: Ex),
{λ T1 IN E}               lambda:         (T1, E: Ex),
{F E}                     apply:          (F, E: Ex),
{D~E}                     binding:        (D, E: Ex),
{REC D~E}                 rec:            (D, E: Ex),
{B1; B2}                  semi:           (B1, B2: Ex),
{LET B IN E}              let:            (B, E: Ex),
                          name:           (N: Name),
{IMPORT N IN E}           import:         (N: Name, E: Ex),
{D→)T0}                   darrow:         (D, T0: Ex),
{N: T}                    sdecl:          (N: Name, T: Ex)
{D1××D2}                  dcross:         (D1, D2: Ex),
{v≠t}                     ev:             (v: V, T: Type)
                          end

type Ex = pointer to ExR;
```

Table 7a: Pascal declarations for the Pebble semantics

```
procedure I(EE: Ex, rho: Binding, typeOnly: boolean, var t: Type, var v: V);
  var
    t0: Type; e0: V;
    tF: Type, f: V;
    tE: Type, e: V;
    B: Ex;
    tx: Type;
    b: Binding;
    Let: Ex;
  begin
  case EE↑.tag of
    . . .
   apply: begin
    I(EE↑.F, rho, tF, f);
    I(EE↑.E, rho, tE, e);
    case tF↑.tag of
     arrow:                                                    {F::t→t0, E:: t}
       if tF↑.domain = tE or HasType(E, tF↑.domain)
       then t0: = tF↑.range else Fail;
     darrow:begin                                              {F::d▶f0, f0#d: = type = F0}
       B: = Bind(EV(tF↑.d, type), EE↑.E));                     {d#type~E = B}
       I(Apply(F, B), rho, tx, v) end;
     else Fail;
     end;
    if not typeOnly then case f↑.tag of
     primitive: begin
       new(e0, bang); e0↑.w: = f↑.w; e0↑.e: = e;               { f≈primitive(w), }
       e0: = Simplify(e0) end;                                 { w!e↝e0 else w!e = e0}
     closure: begin                                            { f≈closure(ρ0, d, E)}
       I(Bind(EV(f↑.domain, type), E), rho, tx, b);            { d#type~E⇒b}
       new(Let, let); let↑.B: = EV(b, f↑.domain); let↑.D: = f↑.body;   { ρ0⊢LET b#d IN E0⇒e0}
       I(Let, f↑.rho, tx, e0) end;
     else begin new(e0, symbolicApply); e0↑.f: = f; e0↑.e: = e end;
     end;
    else e0: = notDone;
    end;
   . . .
   end }
procedure Bind (D, E: Ex): Ex; begin new(Bind, binding); Bind↑.D: = D; Bind↑.E: = E end;
procedureApply (F, E: Ex): Ex; begin new(Apply, apply); Apply↑.F: = F; Apply↑.E: = E end;
procedureEV (v: V, t: Type): Ex; begin new(EV, ev); EV↑.v: = v; EV↑.t: = t end;
```

Table 7b: Pascal code for →**E**

::1 turns $d \bigstar f$ 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.

::2 deals with the case in which typeOf cannot be evaluated immediately. typeE tells us that

typeOf ($x$: int) = int
typeOf ($x$: int × $y$ : real) = int×real,

but

typeOf ($t$: type ×× $x$: $t$) = typeOf!($t$: type★closure($\rho$, $t$: type, $x$: $t$))

because there is no way to compute typeOf ($x$: $t$) without a value for $t$. One might think that there would be some way to obtain

$t$: type★closure($\rho$, $t$: type, $t$))

as the value, but consider

typeOf ($b$: bool ×× IF $b$ THEN $x$: int ELSE $y$: real)

Once we have a pair $E$ in hand, however, it is easy to check whether it has type typeOf!$d \bigstar f$, simply by seeing whether $d \bigstar f \sim E$ type-checks. This expression is checked by :I4-5, which turns $d \bigstar f$ into $d \times t$, much as ::1 does.

*5.3 Type checking* vs *evaluation*

There is a subtle interaction between type-checking and evaluation in Pebble, which is illustrated in Table 6. It is possible to write inference rules which only do type-checking; they must be able to call on the evaluator to evaluate type expressions. It is also possible to write inference rules which only do evaluation, on the assumption that type-checking has already been done. Table 6 repeats the entire set of inference rules twice (except for the auxiliary rules), once with only the parts needed for type-checking, and once with only the parts needed for evaluation. It is important to note that the type-checking rules contain calls on the evaluator, in the form of occurrences of the ⇒ symbol.

Note that most of the rules for ~ and LET are needed for both purposes. This is because these rules set up environments which bind names, and there is no way to tell whether a given name will be needed to evaluate a type expression. In fact, the rules as written in Table 6 are pessimistic; during type-checking it is possible to defer evaluation of the right hand side of a binding until the value of that name is actually needed. As examples (B) and (C) above suggest, this usually happens only when the name denotes a type.

*5.4 Deterministic evaluation*

As we mentioned in § 5.1.2, it is possible to construct a deterministic evaluator from the inference rules. Table 7 gives Pascal declarations for such an evaluator, together with a fragment of the code, that which corresponds to →E. It is interesting to note the close correspondence between the inference rule and the Pascal code, as well as the fact that the code is only about twice as large.

## 6. Conclusion

We have presented both an informal and a formal treatment of the Pebble language, which adds to the typed lambda calculus a systematic treatment of sets of labeled 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 manipulate 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:

Labelled unions or sum types, discussed briefly in § 3.5.

Abbreviations which allow explicit type parameters to be omitted from applications of polymorphic functions.

A sub-type or type inheritance relation, perhaps along the lines suggested by Cardelli.

Assignment, discussed briefly in § 3.6.

Exception-handling, probably 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.

Proof of the soundness of the type-checking, and an exploration of its limitations.

## References

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. and Goguen, J. (1977). Putting theories together to make specifications. *5th Joint International Conference on Artificial Intelligence*, Cambridge, MA, 1045-1058.

Cardelli, L. (1984). A semantics of multiple inheritance. *Lecture Notes in Computer Science* **173**, Springer, 51-68.

Demers, A. and Donahue, J. (1980). Datatypes, parameters and type checking. *7th ACM Symposium on Principles of Programming Languages*, Las Vegas, 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*, Springer.

Lampson, B. and Schmidt, E. (1983). Practical use of a polymorphic applicative language. *10th ACM Symposium on Principles of Programming Languages*, Austin.

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. *Symposium on Lisp and Functional Programming*, Pittsburgh, PA, 243-252.

MacQueen, D., Plotkin, G. and Sethi, R. (1984). An ideal model for recursive polymorphic types. *11th ACM Symposium on Principles of Programming Languages*, Salt Lake City.

MacQueen, D. (1984). Modules for standard ML (draft). In *Polymorphism* (ed. L. Cardelli), Computer Science Dept., Bell Labs., Murray Hill, NJ (privately circulated).

Martin-Lof, P. (1973). An intuitionistic theory of types: Predicative part. In *Logic Colloq. '73* (eds. H.E. Rose and J.C. Shepherdson) North-Holland, 73-118.

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. *JCSS* **17** (3), 348-275.

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 Universitat München.

Plotkin, G. (1981) *A Structural Approach to Operational Semantics.* Computer Science Dept. Report, Aarhus University.

Reynolds, J. (1974) Towards a theory of type structure. *Lecture Notes in Computer Science* **19**, Springer, 408-425.

Reynolds, J. (1983) Types, abstraction and parametric polymorphism, in *Information Processing 83*, North-Holland.

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

Top half of the page is marked with an "a," bottom half with a "b," whole page with neither.