Themes

Simple kernel plus desugaring
for a complete but concise definition.

Declarations as first class objects
to define interfaces and other big things.

Dependent types and type discovery
to make type-checking less obstructive.

Clusters: libraries, inheritance, specialization
to organize the library neatly.

No enforced loss of performance

Participants

Butler Lampson
Rod Burstall
Jim Saxe
John deTreville

Status

Many iterations of design
Several toy implementations
Breadboard implementation underway

Overview

Pebble is a language —
Based on a simple kernel.
With a few essential features:
static type-checking using
symbolic evaluation,
reasoning about equality;
dependent types for
polymorphism,
abstractions;
types as first-class values;
interfaces, modules as first-class values;
exceptions;
side-effects.
Made pleasant for programming by
coercions;
clusters;
discovery functions.
Allowing highly-efficient object code.
With its operational semantics
precisely defined by inference rules
that separate compilation from execution.

Expressions

Names	y, alpha, TYPE	
LET	LET x: INT~3 IN x+5	
Lambda	λ x: INT IN x+5	
Application	mod(i, 5)	
	op."+"(i, 5)	*written* i+5

Binding	y:~3	*for* y: INT~3
Function def	f(y: INT)→(INT):~ y+5	*for* f: (y: INT→INT)~ λ y: INT IN y+5
Selection	(i:~3, j:~5)$i	*for* LET i:~3, j:~5 IN i

Declarations and Bindings

$\boxed{i{:}\mathrm{INT}\sim 3}$ yields the same value as 3, but has type $\boxed{i{:}\mathrm{INT}}$, not INT

binding declaration

The name *i* can be used to refer to the value. Thus

LET i: INT~3 IN i+5

has the value 3+5 or 8

A declaration can be for more than one name, say for *i* and *j*, as in

LET i: INT~3, j: REAL~π IN i+j

which is short for

LET i: INT~3 IN j: REAL~π IN i+j

This binding has type

i: INT × j: REAL

An interface is a declaration, e.g.,

	T	:	TYPE
××	head	:	T→INT
×	tail	:	T→T
×	cons	:	T × INT→T

Interfaces and Implementations

Declaration or interface

```
List :~  T       : TYPE
      xx head    : T→INT
      x  tail    : T→T
      x  cons    : T x INT→T;
```

Binding or implementation

```
Node: TYPE ~ REF RECORD
    next: Node;
    stuff: INT
    END;

NodeList: List ~ (
   T :~ Node;
   head(node: T)→(INT)  :~ node^.stuff;
   tail(node: T)→(T)  :~ node^.next;
   cons(stuff: INT x node: T) → (T) :~ BEGIN
      VAR newNode: T; New(newNode);
      newNode^.next:=node; newNode^.stuff:=stuff;
      RETURN newNode END )
```

Dependent Functions: Polymorphism

Often we want the result type of a function to depend on the argument. Naively:

```
Id:         (T: TYPE →→          (T→T)                       )
Map:        (T: TYPE →→          ((T→T) × LIST T→T))
ZeroArray:  (i: INT  →→          ARRAY [0..i] OF REAL)
```

This doesn't make sense as written, since T and i are not bound by the preceding declaration. But we can take →→ as sugar for a [→] operator that uses a <u>function</u> to compute the result type:

```
Id:         (T: TYPE [→] λ T: TYPE IN (T→T)                       )
Map:        (T: TYPE [→] λ T: TYPE IN ((T→T) × LIST T→T))
ZeroArray:  (i: INT  [→] λ i: INT  IN  ARRAY [0..i] OF REAL)
```

Id and Map can be defined as follows:

```
Id(T: TYPE)(y: T)→(T) :~ y
   Id (INT) (3)  = 3
Map(T: TYPE)(f: (T→T) × l: LIST T)→(LIST T) :~
    IF l=NIL THEN l ELSE cons(f(head l), map(T)(f, tail l))
   Map(INT)(Square, [1, 2, 3]) = [1, 4, 9]
```

Discovering Types

We would like some more sugar to compute the T argument from the y or f argument.

```
  Id(T: TYPE BY ARGTYPE)(y: T)→(T) :~ y
```

after which

```
  Id(3)
```

has ARGTYPE=INT and hence is sugar for

```
  Id(INT)(3)
```

The INT argument is computed by applying the discovery function

```
  λ ARGTYPE: TYPE IN ARGTYPE
```

to the type INT of the argument 3

```
  Map (T: TYPE BY domain firstT ARGTYPE)
      (f: (T→T) × l: LIST T)→(LIST T) :~
      IF l=NIL THEN l ELSE cons(f(head l), map(T)(f, tail l))
```

has the discovery function

```
  λ ARGTYPE: TYPE IN domain firstT ARGTYPE
```

so that

```
  Map(λ i: INT IN i*i, [1, 2, 3])
```

with ARGTYPE=(INT→INT)×LIST INT is sugar for

```
  Map(INT)(λ i: INT IN i*i, [1, 2, 3])
```

Dependent Products: Abstractions

Similarly we may want a pair in which the type of the second depends on the first. Naively:

Any:~	(T: TYPE	xx	T)
Variant:~	(tag: BOOL	xx	IF tag THEN INT ELSE REAL)
List:~	(T: type	xx	head: T→int × tail: T→T × cons: T×int→T)

Again, we can make this work with a function to compute the type of second:

Any:~	(T: TYPE	[×]	λ T: TYPE IN	T)
Variant:~	(tag: BOOL	[×]	λ tag: BOOL IN	IF tag THEN INT ELSE REAL)
List:~	(T: type	[×]	λ T: TYPE IN	head: T→int × tail: T→T × cons: T×int→T)

Examples:

(int, 3) and (real, π) have type Any

(true, 3) and (false, π) have type Variant

(T:~ref Node; head(l: T)→(int):~l^.next; ...) has type List

Modules

Declaration or interface

```
List(U: TYPE)→TYPE :~
        T      : TYPE
   xx   head   : T→U
   x    tail   : T→T
   x    cons   : T × U→T;
```

Binding or implementation

```
Node(U: type)→(TYPE) :~ REF RECORD
  next: Node;
  stuff: U
  END;

NodeList(U: type)→(List) :~ (
  T :~ Node(U);
  head(node: T)→(U) :~ node^.stuff;
  tail(node: T)→(T) :~ node^.next;
  cons(stuff: U × node: T) → (T) :~ BEGIN
    VAR newNode: T; New(newNode);
    newNode^.next:=node; newNode^.stuff:=stuff;
    RETURN newNode END )
```

Abstractions as Parameters

```
List :~  T       : TYPE
     xx  head    : T→INT
     x   tail    : T→T
     x   cons    : T × INT→T;

Reverse(L: List)(y: L$T) → (L$T) :~
  IF y=L$nil THEN y
  ELSE L$nconc(Reverse(L)(L$tail(y), L$head(y)))

SparseMatrix(NL: List(REAL)) → (Matrix) :~
  ....
  VAR n: NL$T;
  ....
  NL$head(NL$tail(n))
  ....
```

Objects

We associate with a type a binding and its associated declaration type, which we call the cluster of the type, writing

```
T WITH B
```

For example,

```
Node with NodeList
```

has type Node and cluster NodeList, where —

```
NodeList: List ~ (
  T :~ Node;
  head(node: T)→(INT) :~ node^.stuff;
  tail(node: T)→(T) :~ node^.next;
  cons(stuff: INT × node: T) → (T) :~ BEGIN
    VAR newNode: T; New(newNode);
    newNode^.next:=node; newNode^.stuff:=stuff;
    RETURN newNode END )
```

Now if y: Node WITH Nodelist, we write

```
y.tail for NodeList$tail(y)
```

or in general

```
E.N for (cluster typeOf E)$N (E)
```

Now we can write

```
Reverse(L: List BY cluster ARGTYPE)(y: L) → (L) :~
  IF y=L$nil THEN y
  ELSE Reverse(y.tail).nconc(y.head)
```