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

<table>
<thead>
<tr>
<th>Names</th>
<th>( y, \alpha, \text{TYPE} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>LET</td>
<td>( \text{LET } x: \text{INT} \rightarrow \text{INT} \text{ IN } x+5 )</td>
</tr>
<tr>
<td>Lambda</td>
<td>( \lambda x: \text{INT} \text{ IN } x+5 )</td>
</tr>
<tr>
<td>Application</td>
<td>( \text{mod}(i, 5) )</td>
</tr>
<tr>
<td></td>
<td>( \text{op.&quot;+&quot;}(i, 5) )</td>
</tr>
<tr>
<td></td>
<td>written ( i+5 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binding</th>
<th>( y: \sim 3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>( f(y: \text{INT} \rightarrow \text{INT}): \sim )</td>
</tr>
<tr>
<td></td>
<td>( \text{for } f: (y: \text{INT} \rightarrow \text{INT}) \sim )</td>
</tr>
<tr>
<td>Selection</td>
<td>( (i: \sim 3, j: \sim 5) )</td>
</tr>
<tr>
<td></td>
<td>( \text{for LET } i: \sim 3, j: \sim 5 )</td>
</tr>
<tr>
<td></td>
<td>( \text{IN i} )</td>
</tr>
</tbody>
</table>
Declarations and Bindings

\[ i : \text{INT}^3 \]

yields the same value as 3, but has type \[ i : \text{INT} \], not \[ \text{INT} \]

binding declaration

The name \( i \) can be used to refer to the value. Thus

\[
\text{LET } i : \text{INT}^3 \text{ IN } i + 5
\]

has the value \( 3 + 5 \) or 8

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

\[
\text{LET } i : \text{INT}^3, j : \text{REAL}^\pi \text{ IN } i + j
\]

which is short for

\[
\text{LET } i : \text{INT}^3 \text{ IN } j : \text{REAL}^\pi \text{ IN } i + j
\]

This binding has type

\[ i : \text{INT} \times j : \text{REAL} \]

An interface is a declaration, e.g.,

\[
T : \text{TYPE}
\]

\[
\times \times \text{head} : T \rightarrow \text{INT}
\]

\[
\times \text{tail} : T \rightarrow T
\]

\[
\times \text{cons} : T \times \text{INT} \rightarrow T
\]
Interfaces and Implementations

Declaration or interface

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

\[
\text{Id: } \quad (T: \text{TYPE} \to T) \\
\text{Map: } \quad (T: \text{TYPE} \to (T \to T) \times \text{LIST} T \to T)) \\
\text{ZeroArray: } (i: \text{INT} \to \text{ARRAY } [0..i] \text{ OF REAL})
\]

This doesn't make sense as written, since \(T\) and \(i\) are not bound by the preceding declaration. But we can take \(\to\) as sugar for a \(\rightarrow\) operator that uses a function to compute the result type:

\[
\text{Id: } \quad (T: \text{TYPE} \to \lambda T: \text{TYPE IN } (T \to T)) \\
\text{Map: } \quad (T: \text{TYPE} \to \lambda T: \text{TYPE IN } (T \to T) \times \text{LIST } T \to T)) \\
\text{ZeroArray: } (i: \text{INT} \to \lambda i: \text{INT IN ARRAY } [0..i] \text{ OF REAL})
\]

\(\text{Id}\) and \(\text{Map}\) can be defined as follows:

\[
\text{Id}(T: \text{TYPE})(y: T \to T) \sim y \\
\text{Id}(\text{INT})(3) = 3 \\
\text{Map}(T: \text{TYPE})(f: (T \to T) \times \text{LIST } T \to (\text{LIST } T)) \sim \\
\text{IF } \text{NIL THEN } \text{ELSE cons}(f(\text{head } l), \text{map}(T)(f, \text{tail } l)) \\
\text{Map}(\text{INT})(\text{Square}, [1, 2, 3]) = [1, t, 3]
\]
Discovering Types

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

\[ \text{Id}(T: \text{TYPE BY ARGTYPE})(y: T \rightarrow T) \sim y \]

after which

\[ \text{Id}(3) \]

has ARGTYPE=INT and hence is sugar for

\[ \text{Id}(\text{INT})(3) \]

The INT argument is computed by applying the discovery function

\[ \lambda \text{ARGTYPE: TYPE IN ARGTYPE} \]

to the type INT of the argument 3

\[ \text{Map}(T: \text{TYPE BY domain firstT ARGTYPE}) \]

\[ (f: (T \rightarrow T) \times 1: \text{LIST T}) \rightarrow (\text{LIST T}) : \sim \]

\[ \text{IF } l = \text{NIL} \text{ THEN } l \text{ ELSE cons(f(head l), map(T)(f, tail l))} \]

has the discovery function

\[ \lambda \text{ARGTYPE: TYPE IN domain firstT ARGTYPE} \]

so that

\[ \text{Map}(\lambda i: \text{INT IN i} \times i, [1, 2, 3]) \]

with ARGTYPE=(INT \rightarrow INT) \times \text{LIST INT} is sugar for

\[ \text{Map}(\text{INT})(\lambda i: \text{INT IN i} \times i, [1, 2, 3]) \]
Dependent Products: Abstractions

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

Naively:

\[
\begin{align*}
\text{Any:} & \quad (T: \text{TYPE} \times T) \\
\text{Variant:} & \quad (\text{tag: BOOL} \times \text{IF tag THEN INT ELSE REAL}) \\
\text{List:} & \quad (T: \text{type} \times (T \rightarrow 
\end{align*}
\]

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

\[
\begin{align*}
\text{Any:} & \quad (T: \text{TYPE} \times \lambda T: \text{TYPE IN} T) \\
\text{Variant:} & \quad (\text{tag: BOOL} \times \lambda \text{tag: BOOL IN} \text{IF tag THEN INT ELSE REAL}) \\
\text{List:} & \quad (T: \text{type} \times \lambda T: \text{TYPE IN} \text{head: } T \rightarrow \text{int} \\
\end{align*}
\]

Examples:

(int, 3) and (real, π) have type Any

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

(\text{list:} \text{ref Node; head(1: T→(int):{\text{null, next; ...) has type List}})
Modules

Declaration or interface

\[\text{List}(U: \text{TYPE}) \rightarrow \text{TYPE} : \sim\]
\[
T : \text{TYPE} \\
\times \text{ head} : T \rightarrow U \\
\times \text{ tail} : T \rightarrow T \\
\times \text{ cons} : T \times U \rightarrow T;\]

Binding or implementation

\[\text{Node}(U: \text{type}) \rightarrow \text{(TYPE)} : \sim \text{REF RECORD}\]

\[
\text{next}: \text{Node}; \\
\text{stuff}: U \\
\text{END};\]

\[\text{NodeList}(U: \text{type}) \rightarrow (\text{List}) : \sim (\]
\[
T : \sim \text{Node}(U); \\
\text{head}(\text{node}: T \rightarrow (U) : \sim \text{node}^\text{.stuff}; \\
\text{tail}(\text{node}: T \rightarrow (T) : \sim \text{node}^\text{.next}; \\
\text{cons}(\text{stuff}: U \times \text{node}: T \rightarrow (T) : \sim \text{BEGIN} \\
\text{VAR} \text{newNode}: T; \text{New(newNode)}; \\
\text{newNode}^\text{.next}:=\text{node}; \text{newNode}^\text{.stuff}:=\text{stuff}; \\
\text{RETURN newNode END } )\]
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$nil THEN y
   ELSE L$conc(Reverse(L)(L$tail(y), L$head(y)))

SparseMatrix(NL: List(REAL)) → (Matrix) ::

.....

VAR n: NL$T;

.....

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

\[ T \text{ WITH } B \]

For example,

Node with NodeList

has type Node and cluster NodeList, where

\[
\text{NodeList: List } \sim (T : \sim \text{ Node;})
\]

\[
\text{head(node: T) } \sim (\text{INT} ) : \sim \text{ node^.stuff;}
\]

\[
\text{tail(node: T) } \sim (T) : \sim \text{ node^.next;}
\]

\[
\text{cons(stuff: INT x node: T) } \rightarrow (T) : \sim \text{ BEGIN}
\]

\[
\text{VAR newNode: T; New(newNode);}
\]

\[
\text{newNode^.next:=node; newNode^.stuff:=stuff;}
\]

\[
\text{RETURN newNode END })
\]

Now if \( y : \text{ Node WITH NodeList, we write} \)

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

or in general

\( E.N \) for \( \text{ (cluster typeOf E)}\$N \) (E)

Now we can write

\[
\text{Reverse(L: List BY cluster ARGM TYPE)(y: L) } \rightarrow (L) : \sim
\]

\[
\text{IF } y=L\$nil \text{ THEN } y
\]

\[
\text{ELSE Reverse(y.tail).nconc(y.head)}
\]