Technical Report

UCAM-CL-TR-31
ISSN 1476-2986

Number 31

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Ponder and its type system

J. Fairbairn

November 1982

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

Ponder and its Type System

Abstract

This note describes the programming language "Ponder", which is designed according to the principles of referencial transparency and "orthogonality" as in [vWijngaarden 75]. Ponder is designed to be simple, being functional with normal order semantics. It is intended for writing large programmes, and to be easily tailored to a particular application. It has a simple but powerful polymorphic type system.

The main objective of this note is to describe the type system of Ponder. As with the whole of the language design, the smallest possible number of primitives is built in to the type system. Hence for example, unions and pairs are not built in, but can be constructed from other primitives.

Section 1: Introduction

Ponder is functional programming language with normal order semantics and a strong polymorphic type checking system. The language is not yet defined in full, and hence this document is open to suggestion, and may not describe the final version of the language.

Ponder is designed with a small set of primitives for declarations and the syntax includes very few "built in" constructs. Normal order semantics were chosen in order that new constructs may be introduced by the programmer, without the problem of unexpected evaluation of arguments, as when attempting to define 'IF', for example. That the language is functional also allows implementations to take advantage of new developments in processor design. Thus no concession is made to conventional machine architectures, since it is hoped that programmes may be efficiently run on (for example) combinator [Clarke 82] or graph reducing machines.

Section 2: Motivation

This section describes the motivation behind the language and its type system, and examines some aspects of other languages. The reader who is unfamiliar with other languages mentioned is asked to bear with me. Some of the notation which occurs is explained later.

Programming languages like Algol68, (and lately Ada), include too much that is built in to the language. This has the effect that they are difficult to remember in full, so that there is a tendency for programmers to only take advantage of a small subset of the language. Furthermore, the excessive size of the definition tends to make implementation difficult, and prone to modification by the implementor in order to make it "efficient".

2.1 Type Checking

Experience has shown that statically checked types in programming languages aid the production of correct programmes in several ways. The most important aspect is that more mistakes in the logic of the programme are detected at compile time, and hence do not need to be found by "debugging". This is particularly

important, since it is often very difficult to test a programme in such a way as to sufficiently 'exercise' all of its individual parts.

A further aspect is that the programmer may take advantage of type checking when making changes to the data structures used within a programme, since the structure of an object is reflected in its type. Thus any incorrect accesses to an object will be reported as type errors, so that any that are missed when the programmer makes a change will be reported by the compiler, rather than lurking to cause a disaster when some part of the programme runs.

2.2

Compile time type checking, then, both protects the programmer from itself*, and provides it with extra facilities which make the writing and maintainance of programmes easier.

Important constructors in data types are STRUCTures (as in Algol68, or records in Pascal), UNIONs (as in Algol68 or as in ML [Gordon 79]), arrays, lists, pairs, and so on, and some form of encapsulation (as in Abstract types in ML). One of the aims of the Ponder type checking system is to provide a mechanism for creating such constructors, and the minimum number of primitives to make them out of.

Unfortunately, the strong type checking as in Algol68 or Pascal is rather too strict, in that it tends to prevent one from doing things which do make sense, as well as things which do not. This tends to make one write the same thing more than once. A commonly quoted example of this is of a routine to sort an array. If one wishes to write a function which given an array and a function saying whether or not two elements are in order, returns a sorted version of the array, one can use the same algorithm regardless of the data type of the elements of the array. In languages such as Pascal or Algol68, this polymorphism (that is to say the ability of a function to work for many "forms" of argument) of the sorting function can not be exploited, and a

*Unfortunately this English usage tends to discriminate against humans. However, the discrimination is less than that against machines suggested by the use of 'him/her'.

different routine must be written for each type of array that is to be sorted. For example, in the notation of Algol68, one might need both a

PROC ([] REAL, PROC (REAL, REAL) BOOL) [] REAL

and a

PROC ([] STRING, PROC (STRING, STRING) BOOL) [] STRING.

This leads to the idea that the type of an object should reflect any polymorphism that is inherent in it. Thus the type of 'sort' should indicate that its first argument is an array of objects of any type, its second is a function taking two objects of that type, and that it returns an array of that type. One might extend the type system of Algol68 to include some kind of type parameters, so that the type of 'sort' might be:

PROC (AMODE M, [] M, PROC (M, M) BOOL) [] M,

the first argument being the type that is to be used throughout.

Type parameters of this nature occur in some other type systems, for example that of Russell [Demers 79], but are accompanied by some philosophical objections: the appearance of a type as a variable suggests that types should be "first class objects", with various operations to split them apart, and to make new ones. If this is the case, then surely types should also be given some sort of type (Indeed in Russell, types do have 'signatures'), but then one can ask why can one not pass the type of the type as a parameter, and so on? Furthermore, the ability of a programme to manipulate its data types in an unpredictable way means that some type checking would be necessary at run time. This is due to the problem that if types may be computed in a perfectly general manner, then the compiler would have to be able to compare functions to see if they always give the same result (which is not generally possible). In Russell types are restricted so that computation of this nature is not done at all, which gives rise to unpleasant effects, such as '**int_array (1+3)**' being a different type from '**int_array (3+1)**'.

One alternative would be to abandon the idea of totally static (compile time) type checking and rely on some dynamic (run time) checking of types. I feel that this is undesirable, since the idea of a type can be either extended to encompass all

programming errors (at one extreme), or restricted to check none (at the other). The first is too general to be really useful, and corresponds for example, to including the fact that divide may not be used with a second argument of zero in its type, and obviously cannot be totally checked before running the programme. The second extreme corresponds to no type checking at all (and equally obviously may be totally checked even before writing the programme!). Thus it is useful to restrict the idea of type errors to that which may be checked for at before running the programme.

My approach has more in common with that in ML, in that type parameters do not really occur. Taking the example of 'sort' again, what one wants to say is that, for all types **'M'**, **'sort'** has type

PROC ([] M, PROC (M, M) BOOL) [] M,

and have the compiler decide what M should be whenever the function is applied. The notation for types in Ponder also has more in common with ML than with Algol68, so that the type of **'sort'** might be written:

∀T. ARRAY [T] × (T × T → BOOL) → ARRAY [T]

2.3 Comparison with ML

The type system of ML answers many of the problems indicated above, but includes more predefined mechanisms than are necessary, and can fail to give a type for some useful functions. In ML a type may include free variables, represented as **'*'**, **'*1'**, **'*2'**, etc, so that the type of **'sort'** in ML is:

(* array) × (* × * → bool) → (* array)

where **'t1 × t2'** is the type of a pair whose left hand element is of type **'t1'**, and whose right hand element is of type **'t2'**, and **'t1 → t2'** is the type of a function requiring its argument to have type **'t1'** and which returns an object of type **'t2'**. My notation differs for type generators (see below for a description), in that for an array of booleans ML has **'(bool array)'** where Ponder has **'ARRAY [BOOL]'**, since I find the prefix notation more readable when there are several applications of generators.

ML is, however, unable to express the types of certain sensible functions. Consider the function:

```
let impossible = λselect. λi. λb.
                    pair (select (i, 1)) (select (b, true))
```

which takes a selector function 'select' (which returns either its first or second argument), and returns a pair constructed of a selected integer, and a selected boolean. In ML parameters to functions are required to have the same type at every application within the function, and the type inference mechanism cannot detect that 'select' must be a polymorphic function, nor can ML types express this fact.

This also has the unpleasant effect that names bound by 'let' are not treated in the same way as names bound by application. Thus:

```
let select = λa. λb. a in
let fs = λi. λb.
        pair (select (i, 1)) (select (b, true))
```

is valid, although 'fs' is equivalent to 'impossible select'.

As will be shown, the type system for Ponder can express a type for such functions, and Ponder preserves the equivalence of binding by declaration and by application. An advantage of this is that type constructors such as pairs and disjoint unions do not have to be built in to the type system, since they may be expressed in terms of simpler constructors. A minor objection to ML types is that the type variables are declared implicitly rather than explicitly. In Ponder all type variables must have explicit declarations, but it is the use of this which provides the extra expressive power.

There is also a slight disadvantage, however. In ML types are arranged so that there is always a most general type for a well typed expression, and so that the compiler can infer what that type is. In Ponder, the type system is somewhat richer, and expressions do not always have a most general type, so of course the compiler cannot be expected to find it. Thus in Ponder the programmer must always specify the types of the parameters of a function. I feel that this is no real loss, since it encourages the programmer to think more about the types of things, and probably improves style. It is also my belief that the programming language should ensure that as much of the

information about a function as possible may be discovered by reading a small amount of the definition of the function. Thus, although it may be difficult at first to get used to function definitions which appear to be cluttered with type information, it is worth while in the end. Programmers used to languages like Algol68 will find that Ponder requires rather less repetition of type information.

Section 3: Semantics

To define the semantics of a programming language in complete formal detail is quite a complicated process, and is beyond the scope of this description. It is, however, desirable that when different people read a Ponder programme they should come to the same conclusion about what it means. To this end, some form of universally understood model is needed (this is clearly impossible). Hence I will describe the semantics of Ponder as transformations into the lambda calculus (see [Hindley 72]).

3.1 Lambda Calculus

I will now give a brief description of λ-calculus, so readers who are already familiar with it should skip to the next heading. Lambda expressions are abstract representations of functions (in the sense of methods of computation). The simplest form of lambda expression is the name (Normally called a variable, but 'variable' has unfortunate connotations of 'variable storage' in computer circles). Here is a lambda expression, then:

days

It does not mean very much, since there is no value associated with '**days**'. Names which occur without defining ocurrences are said to be free. Names may be used as parameters to functions by means of '**λ**', like this:

λparameter_name. body

Where '**body**' may be any form of lambda expression, and '**parameter_name**' is a name for the parameter of the function. It may help to think of '**λx. body**' as "That function **f**, such that **f(x) = body**". I will refer to these as lambda functions. The only other form of expression is the application:

expression_1 expression_2

which means apply '**expression_1**' to '**expression_2**'. The result of applying a lambda function to an argument is the body of the lambda function with all the ocurrences of its parameter replaced by the argument. Hence

(λx. on rainy x) days

evaluates to

on rainy days

The parameter '**x**' having been replaced with the value '**days**'. Note that parentheses are used purely for grouping, and that application associates to the left, so that

the monk ryokan

means the same as

(((the) monk) ryokan)

It is intended that a lambda function should always mean the same thing regardless of the name used for the bound variable, since '**λx. x**' clearly behaves the same as '**λy. y**'. Thus any lambda expression may be replaced by another one which is the same except for the names of bound variables, without change of meaning. When an expression contains nested lambda functions, variables bind to the nearest textually enclosing lambda, so that

λx. (λx.x)

means the same as

λx. (λz.z)

Note that if we allow the simplification of the insides of lambda functions, the process of substitution of a value for a parameter may cause a confusing state to arise. Consider:

λx. ((λy. (λx. x y)) x)

which in all respects behaves the same as

λx. (λz. z x)

in that

λx. ((λy. (λx.x y)) x) p

becomes

((λy. (λx.x y)) p)

and then

λx.x p

which is what you get from

λx. (λz. z x) p

(try it)

If we try to simplify the inside of

λx. ((λy. (λx. x y)) x)

We might try

λx. (λx. x x)

but the last 'x' is clearly meant to be bound to the first one. In these situations the proper thing to do is to change the names so that we can see what is happening, so

λx. ((λy. (λx. x y)) x)

may first be changed to

λx. ((λy. (λz. z y)) x)

and from there we get the correct answer. One final example:

(λx. (λy. x y)) (λy. y y) beep

evaluates to

(λy. (λx. x x) y) beep

then to

(λx. x x) beep

and finally (since 'beep' is unbound, and cannot be evaluated) to

beep beep

Normally descriptions of lambda calculus go on to an extended notation including numbers and so on, but that is unnecessary here.

3.2 Evaluation Order

All objects in Ponder are intended to be capable of representation as lambda expressions. The only other question of semantics which needs answering is "In what order are things evaluated?" In Ponder, function applications are evaluated function first (this is called normal order) as opposed to arguments first, which is usually the case in more conventional languages (and which is called applicative order). To see what this means, consider

(λx. (λy. x)) nice nasty

If we were to evaluate the arguments first, we would have to work out '**nasty**' first, and if '**nasty**' does something nasty, we're in trouble. Normal order evaluation gives us first

(λy. nice) nasty

and then

nice

since '**nasty**' was bound to 'y' which did not appear in the result, which means that the value of '**nasty**' never even gets considered.

Normal order evaluation corresponds approximately to call by name in ALGOL 60, and has a reputation for being inefficient. However, the absence of side effects in functional programming makes it possible to use a scheme of lazy evaluation, in which arguments are evaluated at most once, the first time they are used.

This has the fortunate consequence that it is easy to represent infinite structures, since they are never evaluated in full.

Section 4: Syntax

The syntax of Ponder is intended to be both rich and simple (this is, of course, impossible). The appendix contains a formal description of the syntax of Ponder.

4.1 Symbols

Programmes in Ponder are represented by sequences of symbols. Symbols are divided into four classes, the first being object names. The name of an object should be written in italic letters and numerals, with optional (italic) hyphens to separate individual words within the name. The second class is bold names. These are used for things which are purely syntactic, like the names of data types, and keywords. Bold names are written in bold face roman letters and numerals, with optional (bold face roman) hyphens to separate words within a name.

Unfortunately, current terminals tend not to have italics, and do not preserve the distinctions between hyphens, dashes and minus signs. For this reason it is necessary to adopt the convention that bold names are written in upper case, and names in lower case, using underlines instead of hyphens. hence

feels_sorry_for_himself

is a name, and

ZEN_KOAN

is a bold name. Note that since it is often difficult to distinguish between one and two underlines, consecutive underlines are treated as meaning the same as one.

The third class of symbol is called (for want of a better name) special. Special symbols are either one character, like '(' and ')' (so that '((' is two symbols), or are combinations of the more unusual characters available. The characters which are symbols on their own (and cannot be combined) are:

( ) [ ] { } , ; !

the following characters are the ones normally available in ASCII for making special symbols:

 # $ % & + - * / = ~ ¦ \ < > : . ? @

(It was decided to allow an arbitrary number of these characters to be made into symbols, because it is clear that only allowing one at a time would give too few, and because allowing two at a time seems rather arbitrary and disallows some nice combinations like <=> and =*=. However, it is obviously not a good idea to use long combinations, since it is not easy to read them. For example using both '*******' and '********' would reduce readability.)

Note however that the following symbols already have built in meanings, and may not be used in any other way:

( ) [ ] , ; : . ^ → ∀

When the characters '≐', '↦' and '∀' are not available, they will be represented by '==', '->' and '!' respectively, in which case, these combinations of characters may not be redefined.

The remaining class of symbol is the icon. Icons are symbols whose values and types are determined solely from the text of the symbol. There are currently three types of icon for Ponder. The simplest is the character icon, which is written as an apostrophe, followed by a representation for a particular character. Thus

'a

is a character icon, and stands for the letter a. To enable the representation of non-printing and other difficult characters, character representations include escaped characters, which are written as a further apostrophe followed by another character. Some standard escaped characters are:

'" for "
'' for '
's for space
't for tab

so that

''

is the character icon for apostrophe.

To simplify the inputting of text, there are string icons. A string icon is represented as a quotation mark, followed by a series of character representations, and ending with a further quotation mark. Character representations in strings are the same as in character icons, with the addition that an apostrophe followed by layout is removed up to the next non layout character. This is to allow long strings to be printed on more than one line. Hence

```
"A string icon with a '"line'
 break'" in it"
```

is the string **'A string icon with a "linebreak" in it'**. Note that all layout is removed, so that to insert a space at a line break it is necessary to leave the space before the apostrophe, as in:

```
"Another icon with a line '
 break in it"
```

which means the same as **'"Another icon with a line break in it"'**.

The only other icon is for digit sequences, which are represented by combinations of digits, for instance:

```
42
```

4.2 Declarations

There are two ways of giving a value a name. The first is as the parameter of a function (which is obviously necessary), but the second is by means of the **'LET'** declaration, for example

```
LET three = 3;
```

Which declares the name **'three'** and binds it to the value 3, and to have the type **'INT'**, so that further ocurrences of the name **'three'** mean exactly the same as 3. The general form is

```
LET name = expression; ...
```

Note that the name is declared AFTER the expression, so that, for example

```
LET x = 3;
LET x = x + 1;
x
```

has the value 4, and not infinity.

Semantics

LET declarations may be transformed into lambda expressions:

```
LET name = expression_1; expression_2
```

becomes

```
()name. expression_2) expression_1
```

4.3 Functions

As in lambda calculus, expressions in Ponder may be names, or applications or functions (but see below for more). Names we have seen already, and applications are exactly as described for lambda calculus above.

Functions are represented differently, however. Since Ponder is a typed language, all function parameters must have type definitions, so to simplify the form of functions we have a different syntax.

```
BOOL b → not b
```

is a function, and specifies that the parameter, which is called 'b' is to be of type 'BOOL', and has the value 'not b'. (It may help to pronounce '→' as "returning".) Within the body of a function

```
TYPE name → body
```

'name' may be used where any expression of type 'TYPE' could be used. Hence we may declare named functions as in:

```
LET and = BOOL a → BOOL b →
          IF a
          THEN b
          ELSE false
          FI
```

Semantics

To transform a Ponder function into a lambda function:

TYPE name -> expression

becomes

\name. expression

4.4 Casts

A further form of expression is the cast:

TYPE: old_expression

is an expression which has the type **'TYPE'**, and the same value as **'old_expression'**. Note that this is purely an operation on types, and has no effect on the value of the expression, so the value of the expression must be suitable as an object of type **'TYPE'**. For example

FUNGUS: a_mushroom

is an expression whose type is **'FUNGUS'**, and is valid if every value that **'a_mushroom'** can be is also a **'FUNGUS'**

Semantics

TYPE: expression

becomes

expression

Section 5: Types

Ponder types are similar to the types of MacQueen and Sethi, but are restricted in order to make mechanical type checking possible, and include fewer built in type constructors. The reader is referred to [MacQueen 82] for a more formal treatment of a similar type theory.

Types in Ponder are constructed from other types by means of 'constructors' and 'generators', and by the introduction of named types.

5.1 Function Types

The most obvious constructor is that for functions, which is written '→' (cf Scott [Stoy 77]). Thus if **'LEFT'** and **'RIGHT'** are both types, then **'LEFT → RIGHT'** is the type of a function taking objects of type **'LEFT'** to objects of type **'RIGHT'**. Note that '→' associates to the right, so that **'A → B → C'** means the same as **'A → (B → C)'**, and that as with expressions, parentheses serve only to group things together, so that **'(D → E) → F'** is equivalent to **'(((D → E)) → F)'**, but is probably easier to read.

5.2 Polymorphic Types

One way of introducing a named type is to allow it to be any type at all. This is done by means of the quantifying constructor '∀' (pronounced "for all"), which introduces a name within the rest of a type or expression. Hence **'∀I. I → I'** is a type, which means "for all types 'I', take an object of type 'I', to another object of type 'I'." This is the type of the identity function, which I will now declare as an example:

```
LET identity = ∀I. I i → I: i;
```

The name introduced in this way is known as the variable in which the type is quantified, and things like **'∀T.'** are known as quantifiers. (This type corresponds roughly to the ML type **'* → *'**.) However, the function which takes a selector function as an argument could be written:

```
LET f = (∀S. S → S → S) select →
        INT i → BOOL b →
        pair (select  1 i) (select true b)
```

A further note about binding:

The variable introduced by a quantifier exists as far to the right in its type as is possible, so **'∀T. T → BOOL'** means the same as **'∀T. (T → BOOL)'**, and takes any argument, whereas **'(∀T. T) → BOOL'** demands that its argument has type **'∀T. T'** and hence cannot be expressed in ML. Note that it is mainly in this

respect in which Ponder types differ from the types in ML, in that a quantifier introduces a variable locally within a particular type, whereas in ML all type variables are effectively bound at the outermost level. MacQueen & Sethi's system also allows local type quantifiers.

5.3 Type Generators

The final kind of type constructor is the the type generator. These are declared using declarations similar to:

```
TYPE IDENTITY = ∀I. I → I;
```

This declares a generator 'IDENTITY' which has no parameters, and which means the same as '∀I. I → I'. Generators may also have parameters:

```
TYPE ARROW [LEFT, RIGHT] = LEFT → RIGHT;
```

so that 'ARROW [BOOL, BOOL]' means the same as 'BOOL → BOOL', and 'ARROW [INT, REAL]' is the same as 'INT → REAL', and so on.

Finally Ponder allows types to be recursive, so we can have declarations like

```
RECTYPE INFINITE_LIST [THING] = PAIR [THING, INFINITE_LIST [THING]];
```

which means that 'INFINITE_LIST [INT]' means the same as 'PAIR [INT, PAIR [INT, PAIR [INT, ...]]]' (except that it is easier to see where the recursion goes than deciding what '...' should mean, (or writing it out in full, which would take too long!)).

Note, however, that 'RECTYPE' is intended to declare generators for recursive types rather than recursive functions returning types, hence applications of the generator being declared are restricted to be to the parameters declared within the declaration, and parameters may not be generators. For example:

```
RECTYPE WRONG [T] = WRONG [T → T];
```

is invalid. These restrictions are necessary to make mechanical compile time type checking possible, since without them type generators would be as powerful as the lambda calculus, and hence comparing generated types would be as difficult (undecidable, in fact) as comparing two functions.

Section 6: Relationships Between Ponder Types

It is now necessary to consider which combinations of functions and arguments "make sense" and should be allowed (and if so, what is the type of the result?). One would like to be allowed to do the following:

```
LET identity ≡ ∀I. I i → I: i;
identity something
```

and it is clear that it does not matter what the type of **'something'** is, and also that whatever it is, the type of the expression **'identity something'** is going to be the same. Hence if **'true'** has type **'BOOL'** then **'identity true'** is valid, and also has type **'BOOL'**. If we define 'y' as in:

```
RECTYPE GENY [T] ≡ GENY [T] → (T → T) → T;
LET half_y ≡ ∀T. GENY [T] part_y → (T → T) f → T:
             f (part_y part_y f);
LET y ≡ half_y half_y;
```

then it is not easy to see whether

```
y identity
```

is valid, and if so, what is the type of **'y identity'**?

A more helpful example is

```
LET apply_boolean_operation_to_true ≡ (BOOL → BOOL) op → BOOL:
                                      op true;
apply_boolean_operation_to_true identity
```

since it is clear that although **'op'** is specified as being of type **'BOOL → BOOL'**, **'identity'** will work just as well. The notion being used here is that of the 'generality' of a function. If we require a function of type **'BOOL → BOOL'** in some situation, it is always safe to use the function 'identity' instead, but we can use **'identity'** in situations where a function from 'BOOL' to 'BOOL' will not work, so that **'identity true'** has type 'BOOL' but **'identity 3'** has type integer, and expressions such as **'not 3'** are invalid. We can hence say that the function **'identity'** is more general than any function of type **'BOOL → BOOL'**, or that the type **'∀T. T → T'** is more general than the type **'BOOL → BOOL'**. This relationship is defined more rigorously below.

Thus an object is an acceptable argument to a function if the type of the object is more general than the type of the parameter specified in the declaration of the function.

The non-mathematical reader may like to skip to the end of this section.

For the definition:
The notation

$$\frac{p}{q}$$

means If 'p' is proven, deduce 'q'.

'Tn' are names of types, either in quantifiers or bound at an outer level,

'Kn' are specifiers of types,

'Gn' are generators,

and '$\forall T, \underline{U}.$' is the same as '$\forall T. \forall \underline{U}.$'
($\underline{X}$ means X1, X2, ...)

'$\geq$' is read 'more general (or the same as)', and '$\equiv$' means 'is defined as'

The following rules define the relation:

R1. Reflexivity

$$K1 \geq K1$$

R2. Transitivity

$$\frac{(K1 \geq K2)\ \&\ (K2 \geq K3)}{K1 \geq K3}$$

R3. Instantiation

$$\forall T1.\ G\ [T1] \geq G\ [K]$$

R4. Generalisation

$$\frac{(K \geq G\ [\underline{T}])\ \&\ \underline{T} \text{ not free in } K}{K \geq \forall \underline{T}.\ G\ [\underline{T}]}$$

R5. Function

$$\frac{(K3 \geq K1)\ \&\ (K2 \geq K4)}{K1 \rightarrow K2 \geq K3 \rightarrow K4}$$

R6. Result

$$\forall T.\forall \underline{T1}.G\ [\underline{T1}] \rightarrow G1\ [\underline{T1}, T] \geq \forall \underline{T1}.G\ [\underline{T1}] \rightarrow \forall T.G1\ [\underline{T1}, T]$$

If T is not free in 'G'

R7. Recursion

$$\frac{K1 \geq K2 \Rightarrow G\ [K1] \geq K2\ \&\ K3 = G\ [K3]}{K3 \geq K2}$$

('a => b' is used here to mean 'b' may be deduced from the assumption 'a')

R8. Expansion

$$\frac{G\ [\underline{K}] = K}{G\ [\underline{K}] \geq K\ \&\ K \geq G\ [\underline{K}]}$$

(where '$\underline{K}$' may involve 'K')

R1 simply states that a specifier for a type represents a type which is more general or the same as itself

R2 indicates that the relation is transitive.

R3 notes that a function which works for all types is more general than one which only works for one type.

R4 states that if a type 'K' is more general than some function of 'K1', for all types 'K1', then 'K' is also more general than the generalised version of that function.

R5 is perhaps a little more difficult. At first one might believe that functions requiring more general parameters might me more general. However, this corresponds to a greater restriction on the applicability of the function. For example, if we have:

```
LET u = X x → ...;
LET v = Y y → ...;
```

with 'Y ≥ X' Within the body of 'u' 'x' may be used in any situation where an object as general as 'X' is needed. Similarly 'v' may use its argument as an object of type 'Y'. Hence 'u' will only work if its argument is more general than 'X', and 'v' will only work for arguments more general than 'Y'. But 'Y ≥ X', so all the objects to which 'v' may be applied are also objects to which 'u' may be applied, hence 'u' is more general than 'v'. Thus R5 states that a function which requires less of its argument is more general.

R6 shows that a quantifier which does not appear in the parameter specifier of a function can be moved to be in the result (and so on until it 'drops off the end' if it does not appear in the specifier at all).

R7 allows the comparison of recursive types.

R8 gives a slightly more formal description of the meaning of definition.

6.1 Validity of Application

Using the definition of '≥' we can now specify that an argument of type 'K' is acceptable to a function whose type is 'Ks' if

$$Ks \geq K \rightarrow Kr,$$

for some (see Section 6) Kr.

since this is true if 'K' is more general than the specifier for the parameter of 'Ks' (if it has one).

6.2

The following properties are straightforward consequences of the rules:

Proposition 1

$$\forall T1, T2.\ G\ [T1, T2] \equiv \forall T2, T1.\ G\ [T1, T2]$$

(where 'K1 ≡ K2' means 'K1 ≥ K2 & K2 ≥ K1'). i.e. The order in which quantifiers appear in a type specifier is irrelevant to its meaning.

Proposition 2

$$\forall T,T1.G1[T] \rightarrow G2[T,T1] \equiv \forall T.\ G1[T] \rightarrow \forall T1.\ G2[T,T1]$$

Section 7: Properties

We may now examine the properties of some types.

7.1 Lemma 1:

$$(\forall T.\ T) \geq K \text{ for all } K$$

Proof:

Follows immediately from R3

Hence an object of this type is acceptable as an argument to any function, and no object other than one of this type is acceptable for this type. Thus the only way of creating an object of this type is to declare something like:

```
LET bottom = y identity;
```

(With 'y f' reducing to 'f (y f)'; hence 'y identity' reduces to 'identity (y identity)', and to 'y identity' again, and so on.) which corresponds to a non terminating computation (i.e. there are no values of this type).

7.2 Lemma 2:

If TYPE B = $(\forall T.\ T) \rightarrow B$
Then $K1 \geq B \ \&\ K2 \geq B \ \&\ \ldots \Rightarrow G\ [\underline{K}] \geq B$ for all G [K]

Proof:

By structural induction on G [K]

Case 1:

G [K] = K1
Follows immediately from the assumptions.

Case 2:

G [K] = G1 [K] → G2 [K]

By Lemma 1, $\forall T.\ T \geq G1\ [\underline{K}]$
By induction, $K1 \geq B \ \&\ K2 \geq B \ \&\ \ldots \Rightarrow G2\ [\underline{K}] \geq B$,

Hence by R5,
$K1 \geq B \ \&\ K2 \geq B \ \&\ \ldots \Rightarrow G1\ [\underline{K}] \rightarrow G2\ [\underline{K}] \geq (\forall T.\ T) \rightarrow B$

by R8, R2
$K1 \geq B \ \&\ K2 \geq B \ \&\ \ldots \Rightarrow G1\ [\underline{K}] \rightarrow G2\ [\underline{K}] \geq B$

Case 3:

G [K] = $\forall T.$ G2 [K, T]

By induction
$K1 \geq B \ \&\ K2 \geq B \ \&\ \ldots \ \&\ T \geq B \Rightarrow G2\ [\underline{K}, T] \geq B$

Hence by R2, R4:
$K1 \geq B \ \&\ K2 \geq B \ \&\ \ldots \ \&\ T \geq B \Rightarrow \forall T.\ G2\ [\underline{K}, T] \geq B$

Case 4:

G [K] = G3 [G [K]]

By induction:

K1 ≥ B & K2 ≥ B & ... & G [K] ≥ B => G3 [G [K]] ≥ B

Hence:

K1 ≥ B & K2 ≥ B & ... => (G [K] ≥ B => G3 [G [K]] ≥ B)

Hence by R7:

K1 ≥ B & K2 ≥ B & ... => (G3 [G [K]] ≥ B)

QED.

Note that if 'G' is parameterless, we have

G ≥ B

since '**K**' is empty.

It is clear that, since any object may be applied to an object of type '**∀T.T**', and if it returns at all, it will return an object with the same property, any object is acceptable for '**B**'

Section 8: Result Types

It now remains to answer the question "What is the type of the result of applying some function to some argument when the application is valid?".

We begin by noticing that if there are no quantifiers on the outside of the type of the function, i.e. it is of the form '**Ks → Kr**', then the type returned is clearly the result type '**Kr**'. For instance, if we have:

```
LET true ≙ BOOL:...;
LET bint ≙ BOOL b → INT:
           IF b
           THEN 1
           ELSE 0
           FI;
```

then '**bint (true)**' has type '**INT**'.

We may also notice that if a type is compared with one in which there are free names, the relationship may be dependant upon some constraints on the values of the variables. Hence if we compare 'BOOL → BOOL' with the parameter type of '∀T. (T → T) → T → T' (that is '(T → T)') then 'BOOL → BOOL ≥ (T → T)' is true if both 'BOOL ≥ T' and 'T ≥ BOOL'. We could consider that the result type would be the most general substitution of variables into the result type within these constraints. This seems to work, since for example in

```
LET identity = ∀I. I i → I: i;
LET true = BOOL: ...;
identity true
```

we have 'BOOL ≥ I', and hence the most general type for the result would be 'BOOL'. Unfortunately this does not always make sense. An example is:

```
LET f = ∀T. ((T → T) → T) thing → T: something non obvious;
LET x = (INT → BOOL) ib → BOOL: something less odd;
f x
```

since '(BOOL → INT) → BOOL ≥ (T → T) → T' is true if both 'BOOL ≥ T' and 'INT ≥ T', and it is possible that the only 'T' for which this is satisfied is 'B = (∀T. T) → B'. However, since it is unlikely that such applications were intended, the solution here is to add the condition that an application is only valid if the type which satisfies the constraints is in the set of constraining types. Hence the above example would be rejected.

There are still some situations in which there is no most general type satisfying the conditions. An example of this is:

```
LET f = ∀T. ((T → T) → T) thing → T → T: something odd;
LET x = (A → B) bi → A: something equally odd;
f x
```

In this case we have both 'A ≥ T' and 'T ≥ B', and hence 'A ≥ B', but the result types satisfying the conditions are 'A → A', and 'B → B' and everything between, and none of these types is the most general. In such cases as this the result can be said to have every such type. Thus the type checker should wait until the result of such an application is used before deciding which of the types was intended. (The present implementation of the type checker does not do this, however, since such applications appear to be rare.

Instead it selects one of the possible types by using a similar scheme to that of ML. It is at present not known whether there are any meaningful programmes in which this strategy selects an inappropriate type.)

An alternative approach is found in [MacQueen & Sethi 1982], and is to allow conjunctions of types. This, however tends to result in very complicated types, with little gain for practical applications.

One final problem is that having chosen the result type in this way, it is possible that the bounds on a type variable were in terms of that variable, for example '**A ≥ A → A**', which would be solved by putting '**A = A → A**'. As far as is known, most of the examples in which this situation occurs are nonsensical, and those that make sense may be expressed in ways which avoid the problem (and are usually clearer, for example 'y' above). Given this, it was decided that the type checker should not 'invent' generators as solutions to (in)equations like this, and should reject function applications which produce them (as does ML).

An example of this case is:

```
LET f = ∀A. ((A → A) → A) thing → A: ...;
LET z = ∀T. (T → T) g → T → T: ...;
f z
```

which could be solved by putting '**A = A → A**' in the result, but is instead rejected.

Section 9: Useful Types

The reader may have noticed that I have said nothing about 'primitive' types in Ponder. This is because there are none. Since it is possible to represent any kind of object with an appropriate functional data structure (as in Gedanken [Reynolds 70]), it is not strictly necessary to make any type primitive. For example, we might represent booleans as:

```
TYPE BOOL = ∀B. B → B → B;
LET true = ∀B. B t → B f → B: t;
LET false = ∀B. B t → B f → B: f;
LET if = BOOL b → ∀T. T then_part → T else_part → T:
          b then_part else_part;
```

Note that the definition of 'if' relies on normal order evaluation, which is the order of evaluation defined for Ponder (otherwise the 'then' and 'else' parts would always be evaluated). Thus 'true' is the function which retains its first argument and discards its second, whereas 'false' retains its second.

It would however be desirable to have types such as boolean built in to the implementation, so that programmes may take advantage of properties of the hardware. Unfortunately 'BOOL' as represented above has slightly different properties from booleans, since it includes several objects which fail to terminate if applied. Also 'if' may be given arguments other than (functions which evaluate to) one of the objects 'true' or 'false'. Since this criticism applies to functional data structures in general, it is clearly necessary to provide a mechanism to prohibit unintended applications. The proposed mechanism is to allow the programmer to state that a particular generator is to be sealed, so that it is no longer equivalent to the type which it generates. Thus 'BOOL' might be declared:

9.1 BOOL

```
CAPSULE TYPE BOOL ≐ ∀B. B → B → B;
LET true ≐ BOOL: ∀B. B t → B f → B: t;
LET false ≐ BOOL: ∀B. B t → B f → B: f;
LET if ≐ BOOL b → ∀T. T then_part → T else_part → T:
        b then_part else_part;
SEAL BOOL;
```

which means that 'true' and 'false' are the only objects of type 'BOOL', and that 'if' is the only function which is allowed to take advantage of the representations of 'true' and 'false'. Note that only objects which have been declared to have type 'BOOL' explicitly retain the type 'BOOL' after 'BOOL' is sealed, so that if the declarations included

```
LET spurion ≐ ∀B. B b → B f → B: y i;
```

'spurion' would only have type '∀B. B → B → B'. Conversely, objects which have been stated to be of type 'BOOL' lose their relationship with the representation of 'BOOL' after the 'SEAL', so that one may no longer apply 'true' to anything.

Similarly there are no primitive data structures, and we might declare:

9.2 PAIR

```
CAPSULE TYPE PAIR [T1, T2] = ∀U. (T1 → T2 → U) → U;
LET pair = ∀T1, T2. T1 t1 → T2 t2 → PAIR [T1, T2]:
           ∀U. (T1 → T2 → U) u → U:
           u t1 t2;

LET left = ∀T1, T2. PAIR [T1, T2] p → T1:
           p (∀T1, T2. T1 t1 → T2 t2 → t1);

LET right = ∀T1, T2, PAIR [T1, T2] p → T2:
            p (∀T1, T2. T1 t1 → T2 t2 → t2);
SEAL PAIR;
```

representing 'pair's as functions which take unpacking functions as arguments. Thus for example:

```
left (pair a b)
Reduces to:
pair a b (∀T1, T2. T1 t1 → T2 t2 → t1)
to:
(∀T1, T2. T1 t1 → T2 t2 → t1) a b
to:
(∀T2. T2 t2 → a) b
to:
a
```

The rules for comparing capsules are straightforward: if the names of the two capsules identify with different generators, then they are incomparable. If the names identify with the same generator, then compare the arguments as if comparing the body of the generator with the two sets of arguments substituted.

We now have means of representing 'true' and 'false', and 'PAIR's of things, so it should not be too difficult to represent bit patterns, but what about 'UNION's?

9.3 UNION

```
CAPSULE TYPE UNION [L, R] = ∀E. PAIR [(L → E), (R → E)] → E;
LET inject_l = ∀L. L l → ∀R. UNION [L, R]:
                  ∀E. PAIR [(L → E), (R → E)] p → E:
                  left p l;

LET inject_r = ∀R. R r → ∀L. UNION [L, R]:
                  ∀E. PAIR [(L → E), (R → E)] p → E:
                  right p r;

LET choose = ∀L, R. UNION [L, R] u →
             ∀E. PAIR [(L → E), (R → E)] p →
             u p;
SEAL UNION;
```

The above implements disjoint unions of two types (which may either or both be 'UNION's of other types), as functions which remember an object of one of the two types, and take a 'PAIR' of functions, one for each type. Thus if 'left_thing' has type 'LEFT', 'right_thing' has type 'RIGHT', and 'u_l_r' has type 'UNION [LEFT, RIGHT]',

```
inject_l left_thing
```

creates a '∀R. UNION [LEFT, R]', i.e. a union of 'LEFT' and anything,

```
inject_r right_thing
```

creates a '∀L. UNION [L, RIGHT]', and

```
choose u_l_r
       (pair function_to_apply_if_left
             function_to_apply_if_right)
```

will apply to the element 'function_to_apply_if_left' or 'function_to_apply_if_right' as appropriate.

9.4 LIST

As a final example of the types of functional data structures, here is one version of lists:

```
LET abort = y identity;
CAPSULE RECTYPE LIST [T] = ∀R. (BOOL → T → LIST [T] → R) → R;

LET nil = ∀T. LIST [T]: ∀R. (BOOL → T → LIST [T] → R) f → R:
          f true abort abort;

LET cons = ∀T. T new_head → LIST [T] tail → LIST [T]:
           ∀R. (BOOL → T → LIST [T] → R) f → R:
           f false new_head tail;

LET head = ∀T. LIST [T] l → T:
           l (BOOL null → T h → LIST [T] tail → h);

LET tail = ∀T. LIST [T] l → LIST [T]:
           l (BOOL null → T h → LIST [T] tail → tail);

LET null = ∀T. LIST [T] l → BOOL:
           l (BOOL null → T h → LIST [T] tail → null);

SEAL LIST;
```

In which a list is represented as a function which applies its first argument to a boolean (which is 'true' if the list is 'nil'), the head of the list, and to another list, which is the tail.

Note that **'CAPSULE'** and **'SEAL'** are not brackets, so that the areas where two capsules are open may overlap. However, if you leave the seal off a capsule, you can expect the compiler to complain, since its contents are likely to spill out and ruin your programme.

Section 10: Syntactic Sugar

The language so far described is rather dry, and some things would be a little tedious to do. Hence Ponder includes some mechanisms for introducing new syntactic forms.

10.1 Infix Operators The simplest form of syntactic sugar is to allow infix operators. First the symbol to be used must be declared as an operator, and given a precedence over other operators and a direction for association.

```
PRIORITY n symbol ASSOCIATES direction;
```

Here 'n' is a nonzero digit with '1' being the most binding, 'symbol' is either a bold name other than a keyword, or is a "Special symbol", and 'direction' is either 'LEFT' or 'RIGHT'. Then we can associate an expression with the symbol.

```
INFIX symbol = expression;
```

and

```
TYPEINFIX symbol = NAME_OF_GENERATOR;
```

After such a declaration, the symbol may be used as an infix version of the expression or type. So we might have:

```
PRIORITY 5 - ASSOCIATES LEFT;
PRIORITY 3 * ASSOCIATES RIGHT;
INFIX - = subtract;
INFIX * = times;
```

after which 'a - b - c' means 'subtract (subtract a b) c', 'a * b * c' means 'times (a (times b c))' and 'a * b - c * d' means 'subtract (times a b) (times c d)'.

Further examples:

```
PRIORITY 5 >< ASSOCIATES LEFT;
TYPEINFIX >< = PAIR;

LET or = BOOL a -> BOOL b -> if b true a;
INFIX OR = or;
LET implies = BOOL a -> BOOL b -> (not a) OR b;
```

It would be better if operators were not given priorities, but that their binding power were expressed in relation to other operators, but it is not yet clear what the notation for this should be.

10.2 Prefix Operators

Similarly we can have prefix operators:

```
PREFIX - = minus;
```

so that '-1' means '**minus** 1'.

Prefix operators differ from functions in that they may be overloaded, and that they bind differently. Thus '**minus minus** 1' means '**(minus minus)** 1', whereas '- - 1' means '- (- 1)'.

10.3 Bracketing Operators

The final kind of operator goes round the outside:

```
LET identity = !T. T t -> t;
BRACKET BEGIN END = identity;
BRACKET IF FI = if_thing;
```

so that '**BEGIN expression END**' means the same as '**identity (expression)**'. Note that for a function to be infixed, it must have a type which, when it is applied once, yields another function.

10.4 Overloading

It also would be inconvenient if one had to have a different name for every type of equality function (like '**equals_string_string**' or '**equals_int_int**'), and indeed, programmes would be difficult to read. Hence any kind of operator may be overloaded on the type(s) of its argument(s). Hence we might have:

```
LET equal_bool_bool = BOOL a -> BOOL b -> if a b (not b);
INFIX = = equal_bool_bool;
INFIX = = INT a -> INT b -> ...;
INFIX = = STRING a -> STRING b -> ...;
PREFIX - = INT i -> ...;
PREFIX - = REAL r -> ...;
```

After which all of '"s" = "s"', 'true = true' and '3 = 3' are valid. Note that the overloading is purely syntactic, and that special symbols are not objects, merely syntactic marks.

An overloaded operator application identifies with the most recently declared version of the operator for which an application is valid.

10.5 Pairs

Pairs are however, known more intimately to the compiler. This is to allow "Colateral Declarations" which have proven to be very useful in functional languages. Hence

```
LET a, b = some_pair;
```

means the same as

```
LET Invisible_Name = some_pair;
LET a = left Invisible_Name;
LET b = right Invisible_Name;
```

(Where 'Invisible_Name' is intended to be some name which will not be visible to the rest of the programme).

Although ',' could have been declared as an infix operator for 'pair', it was decided to build this one facility into declarations, since the more general notion of declarations of which this is a special case does not fit readily with the semantics of the lambda calculus, and hence would require something too complicated to be included in Ponder.

Similarly if the argument to a function is a pair, the parts of the pair may be given names, as in

```
LET swap = ∀T1, T2. T1 a, T2 b → b, a;
```

Note that ',' associates to the left, so that 'a, b, c' means 'pair (pair a b) c'. This is hardly important, since the declarations work the same way, so that

```
LET one, two, three = 1, 2, 3;
```

has the obvious effect.

10.6 Recursive Objects

A further form of syntactic sugar is the recursive object declaration:

```
LET_REC factorial = INT n → INT: IF (n <= 1)
                                 THEN 1

                                 ELSE n * (factorial (n - 1))
                                 FI
```

declares the factorial function.

Semantics

LET_REC name = expression_1; expression_2

means the same as:

(λname. expression_2) (y (λname.expression_1))

where

y = λf. (λg. f (g g)) (λg. f (g g))

Section 11: Separate Compilation

In the first versions of the Ponder compiler it will be necessary to allow the programmer to split a programme into several pieces. The mechanism intended for this is similar to that of Algol68c, in that a programme may optionally begin with '**USING "some-definition-file"**' which causes the compiler to read definitions from the file, and may include expressions of the form '**TYPE: ENVIRON "some-other-definition-file"**', which would cause it to output all the preceeding definitions into the file.

Section 12: Standard environment

In order to relieve the programmer of some of the initial definitions, a standard environment file will be provided. This will include definitions of the types '**BOOL, INT, STRING, CHAR,**' and of the generators '**PAIR**', '**LIST**', and possibly some others, together with definitions of some useful infix operators, such as '+', '−', and so on.

Section 13: Example

I now give a complete definition of the construction for '**IF...THEN...ELSE...FI**':

```
CAPSULE TYPE IF_FI [T] = T;
CAPSULE TYPE BOOL = ∀T. T → T → T;
CAPSULE TYPE TE [T1, T2] = PAIR [T1, T2];
BRACKET IF FI = ∀T. IF_FI [T] if_fi → T: if_fi;

PRIORITY 9 THEN ASSOCIATES RIGHT;
INFIX THEN = BOOL b → ∀T. TE [T, T] te → IF_FI [T]:
             BEGIN LET then_part, else_part = te;
                   b then_part else_part
             END;

PRIORITY 9 ELIF ASSOCIATES RIGHT;
INFIX ELIF = ∀T1. T1 then → ∀T2. TE [T2, T2] te → TE [T1, T2]:
             pair then IF te
                       FI;

PRIORITY 9 ELSE ASSOCIATES RIGHT;
INFIX ELSE = ∀T. T1 then → ∀T2. T2 else → TE [T1, T2]:
             pair then else;

SEAL IF_FI;
SEAL BOOL;
SEAL TE;
```

Note the use of 'CAPSULES' to ensure that only objects constructed using the operators may be passed to 'IF ... FI', and the use of pairs to ensure that the various results may have different types, but have some least general type in common.

Section 14: Conclusion

This note has shown that locally quantified polymorphic types with parameterised generators and capsules provide almost all the facilitates required of a type system.

Some useful kinds of types appear at first to be missing from the system, STRUCTures being a notable example. However, much of this can be solved by the use of the mechanism for the overloading of operators, so that a STRUCTure can be represented by a capsule with the appropriate number of 'PAIR's, and overloadable functions for field selectors.

'CAPSULES' do not provide all the facilities of other forms of abstract type, in that everything declared within a **'CAPSULE'** is visible from outside. Hiding is, however already provided within the normal block structure, and is made more palatable with the use of the syntactic sugar for declarations, for example:

```
CAPSULE TYPE BOOL = ∀T. T → T → T;
LET true, false, if = BEGIN LET true = BOOL: ∀T. T t → T f → T: t;
                            LET false = BOOL: ∀T. T t → T f → T: f;
                            LET spurion = ∀T. T t → T f → T:
                                          y identity;
                            LET if = BOOL b → ∀T. PAIR [T, T] te →
                                     T: b (left te) (right te);

                        true, false, if
                      END;
SEAL BOOL;
```

so that **'spurion'** does not even appear in the outside world. This can be improved even more with the use of suitable combining forms for declaration [Milne 76].

The author has implemented a parser and a type checker for Ponder.

Section 15: Acknowledgment

The author wishes to thank his supervisor, Dr M.J.C. Gordon for useful direction, and Dr A.C. Norman for his helpful introduction to functional programming in general.

Appendix: Reference Grammar

The grammar is given as a two level grammar [vWijngaarden 75], but does not attempt to describe the type checking or scope rules.

```
{Metaproductions}
EMPTY:: .

ALPHA:: a; b; c; d; e; f; g; h; i; j; k; l; m;
        n; o; p; q; r; s; t; u; v; w; x; y; z.

NOTION:: ALPHA; NOTION ALPHA.

LEVEL:: i; ii; iii; iiii; iiiii; iiiii i;
        iiiii ii; iiiii iii; iiiii iiii.

PRIO:: prior LEVEL ty.

ASSOC:: left; right.

BRACKET:: name; parenthesis.

{General Hyperrules}
NOTION list: NOTION;
             NOTION, comma symbol, NOTION list.

{Predicates}
provided that PRIO1 greater than or equal PRIO2:
             provided that PRIO1 greater than PRIO2;
             provided that PRIO1 equal PRIO2.

provided that PRIO1 equal PRIO1: true.

provided that prior LEVEL1 LEVEL2 ty
        greater than prior LEVEL3 ty: true.

true: EMPTY.

{Productions}
programme: unit.
```

```
unit: declaration, semicolon symbol, unit;
      representation.

declaration: type dec;
             capsule dec;
             seal capsule;
             operator dec;
             name dec.

operator dec: priority dec;
              infix declaration;
              type infix declaration;
              prefix declaration;
              bracket declaration.

priority dec: priority symbol, integer icon, new operator,
              associates symbol, direction.

infix declaration: infix symbol, new operator,
                   is defined as symbol, representation.

type infix declaration: type infix symbol, new operator,
                        is defined as symbol, bold name symbol.

prefix declaration: prefix symbol, new operator,
                    is defined as symbol, representation.

bracket declaration: bracket symbol, opening BRACKET symbol,
                     closing BRACKET symbol,
                     is defined as symbol, representation.

new operator: PRIO ASSOC operator.

PRIO ASSOC operator: PRIO ASSOC operator symbol;
                     PRIO ASSOC bold name symbol.

type dec: type symbol, generator name,
          is defined as symbol, type;

          rectype symbol, generator name,
          is defined as symbol, type.
```

```
capsule dec: capsule symbol, type dec.

seal capsule: seal symbol, bold name symbol.

generator name: bold name symbol, bound types;
                bold name symbol.

bound types: left bracket symbol, bold name list,
             right bracket symbol.

type: quantified type;
      prior iiiii iiii ty ASSOC map.

quantified type: quantifier, type.

quantifier: for all symbol, bold name list, dot symbol.

PRIO1 ASSOC1 map: solid type;
                  PRIO2 left map, PRIO2 ASSOC1 operator,
                  PRIO2 right map,
                  provided that PRIO1 greater than or equal PRIO2.

solid type: type name; type pack; applied type generator.

applied type generator: bold name symbol,
                        solid type list square pack.

solid type list square pack: left bracket symbol,
                             solid type list,
                             right bracket symbol.

type name: bold name symbol.

type pack: opening parenthesis symbol, type,
           closing parenthesis symbol.

name dec: let symbol, name list,
          is defined as symbol, representation.
```

```
representation: function rep;
                cast;
                prior iiiii iiii ty ASSOC application.

function rep: quantified named map;
              named map.

cast: type, colon symbol, representation.

quantified named map: quantifier, function rep.

named map: named parameters, arrow symbol, representation.

named parameters: typed name;
                  named parameters, comma symbol, typed name.

typed name: solid type, name symbol.

PRIO1 ASSOC1 application:
      prefix application;
      PRIO2 left application, PRIO2 ASSOC1 operator,
      PRIO2 right application,
      provided that PRIO1 greater than or equal PRIO2.

prefix application: operator, prefix application;
                    function application.

function application: expression;
                     function application, expression.

expression: name symbol;
            character icon;
            string icon;
            integer icon;
            bracketed expression.

bracketed expression: opening BRACKET, representation,
                      closing BRACKET.

{end of productions}
```

{The representation of a NOTION-symbol is usually obvious from NOTION. In the case of PRIO ASSOC operator-symbols, however, PRIO and ASSOC are determined by the priority declaration for the symbol in question}

References:

[Clarke 82]: T.J.W. Clarke,
Proceedings of the 1980 Lisp Conference,
The Lisp Company, 1982

[Demers 79]: A.J. Demers & J.E. Donahue,
Revised Report on Russell,
Department of Computer Science Cornell University, 1979

[Gordon 79]: M.J.C. Gordon, A.J. Milner, C.P. Wadsworth,
Edinburgh LCF,
Springer Verlag Lecture Notes in Computer Science No. 78, 1979

[Hindley 72]: Hindley & Seldin,
Introduction to Combinatory Logic,
Cambridge University Press, 1972

[MacQueen 82]: MacQueen & Sethi,
A Semantic Model of Types for Applicative Languages,
Bell Laboratories, 1982

[Milne 76]: R. Milne & C. Strachey,
A Theory of Programming Language Semantics (1.9.3),
Chapman and Hall, 1976

[Reynolds 70]: J.C. Reynolds,
GEDANKEN — A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept,
CACM Vol. 13 No. 5, 1970

[Stoy 77]: J.E. Stoy,
Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory,
MIT Press, 1977

[vWijngaarden 75]: van Wijngaarden et. al.,
The Revised Report on the Algorithmic Language Algol 68,
Springer Verlag, 1975