Technical Report

UCAM-CL-TR-53
ISSN 1476-2986

Number 53

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

A new type-checker for a functional language

Jon Fairbairn

July 1984

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

A New Type-Checker for a Functional Language

Jon Fairbairn
Cambridge University Computer Laboratory,
Corn Exchange Street,
Cambridge CB2 3QG,
United Kingdom.

Abstract

A polymorphic type checker for the functional programming language Ponder [Fairbairn 82] is described. The initial sections give an overview of the syntax of Ponder, and some of the motivation behind the design of the type system. This is followed by a definition of the relation of 'generality' between these types, and of the notion of type-validity of Ponder programmes. An algorithm to determine whether a Ponder programme is type-valid is then presented. The final sections give examples of useful types which may be constructed within the type system, and describe some of the areas in which it is thought to be inadequate.

1. Introduction

This is a description of the type system used in the pure functional programming language Ponder. Ponder was designed as a vehicle for experiments in functional programming style, and to demonstrate that neither purity nor simplicity need be compromised for the sake of utility. To this end it contains no built-in data types or type constructors, no built-in flow control constructs and no assignable variables. Despite this, implementation of Ponder has reached a stage where it is efficient enough for one to consider writing large programmes in it. For example, the performance on Motorola 68000s for first order functions is comparable to that of compiled Lisp.

My intention is not to present any new type-theoretical results, but to describe a practical type-system powerful enough to build many useful types. I describe the syntax of types and an algorithm for a type-checker, and give examples of some constructed types which have been found to be useful. The paper contains neither formal semantics of the type system nor proof of the type-checker, as my main interest has been to develop a practical system. I hope that someone may be inspired to provide both of these.

For the purposes of this paper, I assume that the reason for using a type system is to prevent applications of functions to arguments outside their *intended* domain. To spare the reader the details of Ponder, it is sufficient to know that because there are no built-in types, the underlying objects in Ponder are just untyped λ-functions. It is the intentions of the programmer which determine the domain of a function, since the λ-calculus allows the application of any pair of terms. For example, a λ-function intended to perform addition of two integers represented as by Church [Church 41] would simply do something else if given a λ-function which did not fall within the limits of the representation. The emphasis of this type system is therefore upon intention. If a programmer defines a function which is intended to work on a particular type of datum, it is stated in the programme, and the type checker ensures that the function is never applied to anything of an incompatible type.

On the other hand, one wants to permit the definition of functions which work for *any* type of argument; so the strict typing encountered in 'conventional' programming languages is considered too restrictive. This implies that we need a polymorphic system, such as Milner's for ML [Milner 78]. However, his is too weak to allow the definition of some useful type constructors (see section 5 below for examples), and although one can add other 'built-in' types, this would be contrary to the spirit of Ponder.

Ponder types are similar to the types of MacQueen and Sethi as in [MacQueen 82, 84] but include fewer built-in type constructors, and are restricted in order to make mechanical type-checking possible.

2. Syntax

This section describes the kinds of Ponder expression which are relevant to the discussion of type-checking.

2.1. Expressions

$$\textbf{Expression} = \begin{cases} \textbf{Name} & \text{Variable} \\ \textbf{Expression}_1\ \textbf{Expression}_2 & \text{Application} \\ \textbf{Type Name} \rightarrow \textbf{Expression} & \text{Abstraction} \\ \textbf{Type} : \textbf{Expression} & \text{Cast} \\ \forall V.\textbf{Expression} & \text{Quantified expression} \end{cases}$$

2.1.1. Grouping

Parentheses are used merely to achieve a particular binding, but square brackets are used around lists of arguments to type generators.

2.1.2. Function Application

The syntax of function application is the same in Ponder as in the λ-calculus: $f\ x$ means 'apply f to x'. Application associates left, so $f\ x\ y$ means $(f\ x)\ y$

2.1.3. Abstraction

Ponder uses a compact method of specifying the name and type of the bound variable of a function:

Type Name → Body

is a function with bound variable **Name** which is required to have type **Type**. The equivalent λ-function would be λ**Name . Body.**

For example

Int i → plus i 1

is a function taking an argument of type *Int* called *i*, and returning the value of *plus* applied to *i* applied to 1 (i.e. $\lambda i.plus\ i\ 1$ in the λ-calculus).

2.1.4. Casts

A cast is an expression which is asserted by the programmer to have a particular type:

Type : Expression

means that **Expression** must have type **Type.**

2.1.5. Quantified Expressions

An expression may be preceded by a type quantifier:

$\forall V$**.Expression**

This declares the type variable V in the expression and states that the expression must work for any V.

2.2. Constructing Types

Ponder types are constructed from function types, quantified types and type generators:

$$\mathbf{Type} = \begin{cases} V & \text{Type variables} \\ \mathbf{Type} \rightarrow \mathbf{Type} & \text{Functions} \\ \forall V.\mathbf{Type} & \text{Quantified types} \\ G[\mathbf{Type}, \ldots, \mathbf{Type}] & \text{Generators} \end{cases}$$

'Generators' are user defined parameterised type constructors. A further facility is the capsule, which provides a very simple kind of encapsulation of types.

Note that there are no built-in types at all. This is unimportant since parameterless capsules which behave like built-in types (such as *Integer*) can be constructed. Capsules with parameters perform a similar function to that of built-in type constructors.

2.2.1. Naming

For clarity I will give types names which are either a single upper-case letter (e.g. T), or are words beginning with an upper-case letter followed by letters, hyphens and possibly with a subscript (like T, T_1, *Long-name*, ...).

2.2.2. Function Types

The most primitive kind of type is that of the function from one type to another, which is written using '$\rightarrow$'. Thus if *Parameter* and *Result* are both types, then *Parameter* $\rightarrow$ *Result* is the type of a function taking objects of type *Parameter* to objects of type *Result*. Note that $\rightarrow$ associates to the right, so that $A \rightarrow B \rightarrow C$ means the same as $A \rightarrow (B \rightarrow C)$.

2.2.3. Quantifiers

The universal quantifier, $\forall$, (pronounced 'for all'), introduces a name within the rest of a type or expression. Hence $\forall I.I \rightarrow I$ 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 $\lambda x\,.\,x$.

A note about binding: the scope of a variable introduced by a quantifier extends as far to the right as possible, but is limited by parentheses, so $\forall T.T \rightarrow \mathit{Bool}$ means the same as $\forall T.(T \rightarrow \mathit{Bool})$, and takes any argument, whereas $(\forall T.T) \rightarrow \mathit{Bool}$ demands that its argument has type $\forall T.T$ (and hence could not be expressed in Milner's type system). For the sake of convenience, $\forall T.\forall U \ldots$ may be written $\forall T, U \ldots$

2.2.4. Type Generators

The final kind of type constructor is the type generator, which is a user defined operator which generates a type, for example:

$$\mathit{TYPE\ Identity} \triangleq \forall I.I \rightarrow I;$$

This declares ($\triangleq$ means 'is defined as') a generator *Identity* which means the same as $\forall I.I \rightarrow I$.

Generators may also have parameters:

$$TYPE\ Arrow\ [Left, Right] \triangleq Left \rightarrow Right$$

so that $Arrow\ [Bool, Bool]$ means the same as $Bool \rightarrow Bool$, and $Arrow\ [Int, Real]$ is the same as $Int \rightarrow Real$, and so on. For example, we might define pair types as

$$TYPE\ Pair\ [L, R] \triangleq \forall Res.(L \rightarrow R \rightarrow Res) \rightarrow Res$$

Note that this is an example of a type which is not expressible in ML. (This type is described in more detail in the section 'Representing Objects' below.)

Finally Ponder allows types to be recursive (but in a restricted way); so we can have declarations like

$$RECTYPE\ Infinite\text{-}list\ [T] \triangleq Pair\ [T, Infinite\text{-}list\ [T]]$$

which means that $Infinite\text{-}list\ [Int]$ means the same as $Pair\ [Int, Pair\ [Int, Pair\ [Int, \ldots]]]$.

Recursive generators are restricted to ensure that all types generated are finite cycles rather than infinite trees. One way of looking at this restriction is to consider it in terms of a *μ-operator* for type generators. μ binds a type variable (*not* a generator) and finds a fixed point of a type generator, so that if we were to allow type definitions like

$$TYPE\ G\ [X] \triangleq \mu T.T \rightarrow X$$

then G would be the same as if it were defined as in

$$RECTYPE\ G\ [X] \triangleq G\ [X] \rightarrow X.$$

Recursive types in Ponder are just those types constructible with the *μ-operator*. Hence in a definition

$$RECTYPE\ G\ [T_1, \ldots, T_n] \triangleq Body$$

all applications of G within *Body* must be to *exactly* the parameters $[T_1, \ldots, T_n]$. For example:

$$RECTYPE\ Invalid\ [T] \triangleq Invalid\ [(T \rightarrow T)]$$

is invalid. This restriction is necessary to make mechanical compile-time type checking possible, since without it type generators would be equivalent to the λ-calculus, and hence type-checking would be undecidable, since comparison of types would have the same complexity as computation of the equality of functions.

2.2.5. Capsules

Normally, two differently named definitions of the same type are equivalent; i.e. are indistinguishable in use (the exact rules for equivalence are given below). For example, if

$$TYPE\ A \triangleq \forall T_1.T_1$$

and

$$TYPE\ B \triangleq \forall T_1.\forall T_2.T_2$$

A and B may be used interchangeably since they define equivalent types. This is not always what is wanted, so capsules are provided as a means of destroying the referential transparency of a particular name. Capsules are the same as generators, except that the structure of the declared type is hidden for the purposes of type-checking. This means that, for example

$$CAPSULE\text{-}TYPE\ P \triangleq Int$$

is considered to be a different type from *Int*. Thus capsules obey 'name equivalence' (as do abstract types in ML), whereas types in general obey the rules of equivalence given below, in a similar way to the structural equivalence of types in Algol 68 [van Wijngaarden 75].

3. Type Checking

This section introduces the rules to which a Ponder programme must adhere.

Although it is not mentioned above, Ponder allows the definition of *overloaded* operators (operators for which the meaning is dependent upon the types of the arguments). This means that it is impossible to infer the type of an expression unless the types of all the variables are known. However, the syntax ensures that all variables are declared with a particular type.

On the other hand, it would be tedious to have to give the result types of every function application, because in most cases it is possible to infer the result type from the types of the function and argument. Hence the Ponder type-checker must be capable of determining whether an expression is type correct, given the types of all the variables present.

In order to describe this, I need to give a definition of the relationship of generality between types. The relation $T_1 \geq T_2$ is intended to mean that any object of type T_1 may validly be used in any situation where an object of type T_2 may validly be used. Thus, if *Identity* is as defined above, $Identity \geq (Int \rightarrow Int)$, since the any object of type *Identity* (they are all equivalent to $\lambda x.x$) may safely be used where an object of type $Int \rightarrow Int$ may be used.

As programmes contain sub-expressions with type variables defined at an outer level, it is necessary to consider the possibility of comparison of types containing free variables. For example, within the body of a function of type $\forall V.V \rightarrow \ldots$, the parameter of the function will be of type V. Free type variables like this can be treated as constant types, because there is no information available about the type that they represent.

Capsules with no arguments are treated in exactly the same way as type variables bound at an outer level. Capsules with arguments require slightly more complex treatment, since we want to ensure that applications of the same capsule to different arguments compare in a friendly way. If *Arrow* were a capsule, we would still want $Arrow\ [Int, Identity] \geq Arrow\ [Int, (Int \rightarrow Int)]$. This is achieved by observing that to compare capsules, one first discovers whether they are the same capsule (if not, then

they are incomparable), and then compares the types resulting from the application of the capsule to its two sets of arguments.

3.1. The Relation of generality between types

Rules R1 to R8 below define the relation $\geq$. V_n are type variables, T_n are arbitrary types (possibly with free variables), G_n are generators, Γ stands for a set of assumptions each of which is of the form $T_1 \geq T_2$ or $G[\ldots] \triangleq T$ and $\geq$ is as above.

Reflexivity

$$\Gamma \vdash T \geq T \qquad R1$$

This means that from any set of assumptions Γ and type T we can deduce that $T \geq T$; i.e. any type is at least as general as itself.

Transitivity

$$\frac{\Gamma_1 \vdash T_1 \geq T_2, \quad \Gamma_2 \vdash T_2 \geq T_3}{\Gamma_1 \cup \Gamma_2 \vdash T_1 \geq T_3} \qquad R2$$

In rules such as this, assumptions are written above the line and conclusions below it. This rule may be read as 'If we can prove from Γ_1 that $T_1 \geq T_2$ and from Γ_2 that $T_2 \geq T_3$, then we can prove from the union of Γ_1 and Γ_2 that $T_1 \geq T_3$.'

Instantiation

$$\Gamma \vdash \forall V.G[V] \geq G[T] \qquad R3$$

A function which works for all types is more general than one which only works for one type.

Generalisation

$$\frac{\Gamma \vdash T \geq G[V], \; V \text{ not free in } T}{\Gamma \vdash T \geq \forall V.G[V]} \qquad R4$$

If a type T is more general than a type parameterised on V, regardless of the value of V, then T is also more general than the generalised version of that function.

Function

$$\frac{\Gamma_1 \vdash T_3 \geq T_1, \quad \Gamma_2 \vdash T_2 \geq T_4}{\Gamma_1 \cup \Gamma_2 \vdash T_1 \to T_2 \geq T_3 \to T_4} \qquad R5$$

A function which requires less of its argument is more general.

Result

$$\frac{V \text{ not free in } T}{\Gamma \vdash \forall V.T \to G[V] \geq T \to \forall V.G[V]} \qquad R6$$

A quantifier which does not appear in the parameter specifier of a function can be moved to the result.

Recursion

$$\frac{\Gamma_1, T_1 \geq T_2 \vdash G[T_1] \geq T_2, \quad \Gamma_2 \vdash G[T_3] \triangleq T_3}{\Gamma_1 \cup \Gamma_2 \vdash T_3 \geq T_2} \qquad R7$$

this rule allows the comparison of recursive types.

Expansion

$$\frac{\Gamma \vdash G[T_1, \ldots T_n] \triangleq T}{\Gamma \vdash T \geq G[T_1, \ldots T_n]} \qquad R8a$$

$$\frac{\Gamma \vdash G[T_1, \ldots T_n] \triangleq T}{\Gamma \vdash G[T_1, \ldots T_n] \geq T} \qquad R8b$$

(where $T_1, \ldots T_n$ may involve T); this gives the meaning of definition.

3.2. Properties

The type checker makes tacit use of some straightforward consequences of these rules including the following:

$$\forall V_1, V_2.T \equiv \forall V_2, V_1.T$$

(where $T_1 \equiv T_2$ means $T_1 \geq T_2$, $T_2 \geq T_1$, T_1 is equivalent to T_2), i.e. the order in which quantifiers appear in a type is irrelevant to its meaning.

$$\forall V_1, \ldots V_n, V.T \to G[V] \equiv \forall V_1 \ldots V_n.T \to \forall V.G[V]$$

Hence types may be equivalent even when they have different textual representations.

$$\Gamma \vdash (\forall V.V) \geq T \text{ for all } T$$

3.3. Rules for Type-Validity of Expressions

This section presents the rules to which valid Ponder programmes must conform. In general a programme will consist of a 'casted' expression, the type of which is determined by the environment in which the programme is intended to run. It is also meaningful to say that a programme without a cast at the outermost level must be type-valid—this means that the programme must have at least one type.

A programme p is type-valid if a statement of the form $T : p$ for some T may be proven within the following rules.

The notation $T : e$ means that e has the type T, and Γ is a set of assumptions as before but may also include assumptions of the form $T : e$.

Application

$$\frac{\Gamma_1 \vdash (T_1 \rightarrow T_2) : e_1,\ \Gamma_2 \vdash T_1 : e_2}{\Gamma_1 \cup \Gamma_2 \vdash T_2 : e_1\ e_2} \qquad V1$$

Function

$$\frac{\Gamma,\ T_1 : v \vdash T_2 : e,\ v \text{ not free in } \Gamma}{\Gamma \vdash T_1 \rightarrow T_2 : (T_1 v \rightarrow e)} \qquad V2$$

Cast

$$\frac{\Gamma \vdash T_1 : e}{\Gamma \vdash T_1 : (T_1 : e)} \qquad V3$$

Generalisation

$$\frac{\Gamma \vdash G[T] : e,\ T \text{ not free in } \Gamma}{\Gamma \vdash \forall V.G[V] : \forall V.e} \qquad V4$$

Restriction

$$\frac{\Gamma_1 \vdash T_1 : e,\ \ \Gamma_2 \vdash T_1 \geq T_2}{\Gamma_1 \cup \Gamma_2 \vdash T_2 : e} \qquad V5$$

4. The Type Checking Algorithms

Having established the meaning of type-validity, I now describe an algorithm for type-checking Ponder programmes. The algorithm is presented as a function *type-check*, followed by definitions of subsidiary functions. For the sake of clarity, the mechanism which deals with recursion is described separately.

In what follows, V are type variables, T are types, and e are expressions.

4.1. *type-check*

The type checker (*type-check*) takes as argument a triple $(\mathbf{A}, \tau, e)$, where $\mathbf{A}$ is a set of assumptions about type variables, each element of which is one of

(a) Fixed V,
(b) $V \leq T$,
(c) $T \leq V$,

e is a Ponder expression and τ is a set of typings of the form $T : e$.

The result of *type-check* $(\mathbf{A}, \tau, e)$ is a pair $(\mathbf{A}', T)$, where T is a type and $\mathbf{A}'$ is either a set of type assumptions (as defined above), or 'Fail'. If $\mathbf{A}'$ is a set of assumptions, then T is the type of e, when interpreted within those assumptions, otherwise we conclude that e is invalid. In other words, *type-check* $(\mathbf{A}, \tau, e) = (\mathbf{A}', T)$ implies that $\mathbf{A} \cup \tau \vdash T : e$.

The type checker requires two subsidiary functions *valid*, and $\boxed{\leq}$; *valid* checks a set of constraints and either returns the same set of constraints or fails, and $T_1 \boxed{\leq} T_2$ is the set of assumptions needed to show that $T_1 \leq T_2$.

The definition of *type-check* is given as a case analysis of possible expressions, together with a short description of the idea underlying each particular case.

Variables

The type of a variable is given by its environment:

$$\textit{type-check}\ (\mathbf{A}, \tau, v) = (\mathbf{A}, T),\ \text{where}\ (T : v) \in \tau;$$

Application

To check an application, check the function and argument, and then calculate the result type:

$$\begin{aligned}\textit{type-check}\ (\mathbf{A}, \tau, e_1\ e_2) = (&\textit{valid}\ (\mathbf{A}_1 \cup \mathbf{A}_2 \cup T_1 \boxed{\geq} (T_2 \to V_r)), V_r),\\ &\text{where}\ (\mathbf{A}_1, T_1) = \textit{type-check}\ (\mathbf{A}, \tau, e_1)\\ &\text{and}\ (\mathbf{A}_2, T_2) = \textit{type-check}\ (\mathbf{A}, \tau, e_2)\\ &\text{and}\ V_r \text{ is not free in } \mathbf{A}_1, \mathbf{A}_2, T_1 \text{ or } T_2\end{aligned}$$

Function

Type-check the body of the function given that the parameter has the stated type; the result type is a function from the parameter type to the type of the body:

$$\begin{aligned}\textit{type-check}\ (\mathbf{A}, \tau, T_v v \to e) = &(\mathbf{A}', T_v \to T_e),\\ &\text{where}\ (\mathbf{A}', T_e) = \textit{type-check}\ (\mathbf{A}, \tau \cup \{T_v : v\}, e)\end{aligned}$$

Casts

The type of a cast expression is the type given in the cast, but we must check to see that the type of the expression $\geq$ that type:

$$\begin{aligned}\textit{type-check}\ (\mathbf{A}, \tau, T : e) = (A_2, T),\ &\text{where}\ \mathbf{A}_2 = \textit{valid}\ (\mathbf{A}_1 \cup T_1 \boxed{\geq} T)\\ &\text{where}\ (\mathbf{A}_1, T_1) = \textit{type-check}\ (\mathbf{A}, \tau, e)\end{aligned}$$

Quantified Expressions

The body of a quantified expression is checked given that the type variable is fixed:

$$\begin{aligned}\textit{type-check}\ (\mathbf{A}, \tau, \forall V.e) = &(\mathbf{A}' - \{\text{Fixed } V\}, \forall V.T_e),\\ &\text{where}\ (\mathbf{A}', T_e) = \textit{type-check}\ (\mathbf{A} \cup \{\text{Fixed } V\}, \tau, e)\end{aligned}$$

4.1.1. $\boxed{\geq}$

$\boxed{\geq}$ is an infix operation between two types. $T_1 \boxed{\geq} T_2$ is either Fail or a set of assumptions $\mathbf{A}$, such that $\mathbf{A} \vdash T_1 \geq T_2$.

The following rewriting rules define $\boxed{\geq}$ case by case ($a \Rightarrow b, c$ means reduce a to b, otherwise try c):

$$V \boxed{\geq} \forall V_1.T \Rightarrow \{V \geq T, \text{ Fixed } V_1\}, \quad C0$$
$$V \boxed{\geq} T \Rightarrow \{V \geq T\}, \quad C1$$
$$\forall V.T_1 \boxed{\geq} T_2 \Rightarrow T_1 \boxed{\geq} T_2, \quad C2$$

This corresponds to *R3*

$$T_1 \to T_2 \boxed{\geq} V \Rightarrow \{T_1 \to T_2 \geq V\}, \quad C3$$
$$T_1 \to T_2 \boxed{\geq} \forall V.T_3 \Rightarrow (T_1 \to T_2 \boxed{\geq} T_3) \cup \{\text{Fixed } V\}, \quad C4$$

the 'Fixed' represents the fact that V must be free in T_3 as in *R4*

$$T_1 \to T_2 \boxed{\geq} T_3 \to T_4 \Rightarrow (T_3 \boxed{\geq} T_1) \cup (T_2 \boxed{\geq} T_4) \quad C5$$

4.2. Checking Assumption Sets

I now give the rules for *valid*. *valid* checks a set of assumptions for consistency, returning Fail if the set contains contradictory assumptions, and a simplified set otherwise. A is a set of assumptions as for *type-check*, and $\boxed{\vee}$ is defined below.

$$\textit{valid}\ (\{\text{Fail}\} \cup \mathbf{A}) \Rightarrow \text{Fail}, \quad S1$$
$$\textit{valid}\ (\{V \leq T_1, V \text{ Fixed}\} \cup \mathbf{A}) \Rightarrow \text{Fail}, \quad S2$$
$$\textit{valid}\ (\{V \geq T_1, V \text{ Fixed}\} \cup \mathbf{A}) \Rightarrow \text{Fail}, \quad S3$$
$$\textit{valid}\ (\{V \leq T_1, V \leq T_2\} \cup \mathbf{A}) \Rightarrow \textit{valid}\ (\{V \leq T_1\} \cup (T_1 \boxed{\vee} T_2) \cup \mathbf{A}), \quad S4$$
$$\textit{valid}\ (\{V \leq T_1, V \geq T_2\} \cup \mathbf{A}) \Rightarrow \begin{cases} \text{Fail if } T_1 \boxed{\geq} T_2 \text{ is Fail,} \\ \{V \geq T_2\} \cup \textit{valid}\ (\mathbf{A} \cup \{V \leq T_1\}) \text{ Otherwise} \end{cases} \quad S5$$

4.2.1. $\boxed{\vee}$

This operation performs a similar function to unification. $T_1 \boxed{\vee} T_2$ is the set of restrictions on variables in T_1 and T_2 needed to find a T_3 such that $T_3 \leq T_1$ and $T_3 \leq T_2$.

$$V \boxed{\vee} T \Rightarrow \{V \boxed{\geq} T, T \boxed{\geq} V\},$$
$$\forall V.T_1 \boxed{\vee} T_2 \Rightarrow T_1 \boxed{\vee} T_2,$$
$$T_1 \to T_2 \boxed{\vee} V \Rightarrow \{T_1 \to T_2 \boxed{\geq} V, V \boxed{\geq} T_1 \to T_2\},$$
$$T_1 \to T_2 \boxed{\vee} \forall V.T_3 \Rightarrow T_1 \to T_2 \boxed{\vee} T_3,$$
$$T_1 \to T_2 \boxed{\vee} T_3 \to T_4 \Rightarrow T_2 \boxed{\vee} T_4$$

4.2.2. Examples

If f and x have been declared with types $(Int \to Int) \to Bool$ and $(\forall T.T \to T)$ respectively, then *type-check* $(\{\}, \{(Int \to Int) \to Bool : f, (\forall T.T \to T) : x\}, f\ x)$ results in checking f and x, which in turn results in

$$(valid\ ((Int \to Int) \to Bool \boxed{\geq} (\forall T.T \to T) \to V_r), V_r)$$

taking

$$(Int \to Int) \to Bool \boxed{\geq} (\forall T.T \to T) \to V_r$$

we get

$$((\forall T.T \to T) \boxed{\geq} Int \to Int) \cup (Bool \boxed{\geq} V_r) \qquad \text{By } C5$$

$$(\forall T.T \to T \boxed{\geq} Int \to Int) \cup \{V_r \leq Bool\} \qquad \text{By } C1$$

$$((T \to T) \boxed{\geq} (Int \to Int)) \cup \{V_r \leq Bool\} \qquad \text{By } C2$$

$$(Int \boxed{\geq} T) \cup (T \boxed{\geq} Int) \cup \{V_r \leq Bool\} \qquad \text{By } C5$$

$$\{V_r \leq Bool, T \geq Int, T \leq Int\} \qquad \text{By } C1 \text{ (twice)}$$

valid checks $Int \geq Int$ (which is true), so the answer is

$$(\{V_r \leq Bool, T \geq Int, T \leq Int\}, V_r)$$

Which means that $f\ x$ has type V_r, provided that $V_r \leq Bool$.

Suppose that the argument and parameter types had been the other way round. The initial tests would follow the same course, but

$$(\forall T.T \to T) \to Bool \boxed{\geq} (Int \to Int) \to V_r$$

would reduce like this:

$$(Int \to Int \boxed{\geq} \forall T.T \to T) \cup Bool \boxed{\geq} V_r \qquad \text{By } C5$$

$$(Int \to Int \boxed{\geq} \forall T.T \to T) \cup \{V_r \leq Bool\} \qquad \text{By } C1$$

$$(Int \to Int \boxed{\geq} T \to T) \cup \{V_r \leq Bool, T \text{ Fixed}\} \qquad \text{By } C4$$

$$(T \boxed{\geq} Int) \cup (Int \boxed{\geq} T) \cup \{V_r \leq Bool, T \text{ Fixed}\} \qquad \text{By } C5$$

$$\{V_r \leq Bool, T \text{ Fixed}, T \leq Int, T \geq Int\}$$

but this time *valid* would produce Fail (By $S2$).

4.3. Recursive generators

In the absence of recursive type generators it is clear that rules $C1$–5 will produce a finite set of assumptions, since each rule involves a reduction in size of the comparands.

Recursive generators have the effect that types are no longer finite, and so recursion on their structure may not terminate. The restrictions Ponder imposes on recursive generators ensure that the expansions of recursive calls to type generators are equivalent to the original application;

$$G[T_1, \ldots, T_n] \triangleq \ldots G[T_1, \ldots, T_n] \ldots$$
$$\Uparrow$$

so the application of G marked with $\Uparrow$ will be equivalent to the initial application of G.

Since the texts of recursive generators are finite, it is clear that any infinite series of comparisons which could arrive must be a cycle. Hence it suffices to introduce a memory into the algorithm, and not perform any comparison twice (the assumptions generated must be the same as some which have occurred already).

Note that the same mechanism is useful for *valid*, in order to avoid checking the same thing twice.

4.4. Optimisations

The description of the algorithm above is simplified. In fact, it is desirable that a type error should be detected as soon as possible (the overloading mechanism of Ponder relies on type-checks which fail).

A number of optimisations to the algorithm are therefore desirable. Instead of computing the sets of assumptions and then checking for consistency at the end it is useful to pass the set as computed so far to succeeding comparisons, and use a special function to insert new elements in the set. This means that inconsistencies of the form $\{V \leq T, V \text{ Fixed}\}$ are discovered immediately, and the comparison may stop. To speed the process of insertion, it is useful to sort the assumptions by variable, and to duplicate cases where two variables are compared, so that the relationship is keyed by both variables.

A further optimisation arises from the observation that it is not necessary to perform an expansion when comparing type generators which are the same. A more efficient approach is to compare only the arguments of the two generators. However, it is necessary to pre-calculate the directions which comparisons will take. For example if $F[A, B] \triangleq A \rightarrow B$, then for $F[A_1, B_1] \geq F[A_2, B_2]$ we need $B_1 \geq B_2$, but $A_2 \geq A_1$. In a comparison $G[T_{1L}, \ldots T_{nL}] \boxed{\geq} G[T_{1R}, \ldots T_{2R}]$, each T_{iL} will be compared with the corresponding parameter T_{iR}, either by $T_{iL} \boxed{\geq} T_{iR}$, or by $T_{iR} \boxed{\geq} T_{iL}$ or both. It is a simple matter to pass over the definition of a generator and classify each parameter according to which way it will be compared.

5. Representing Objects

Because there are no built-in functions or data-types, objects in Ponder are represented as functional data-structures. In this section I give the definitions of a few types which one would normally expect to be built in.

None of the examples is particularly complex; I merely wish to show that the type system is strong enough for them, and add that it is possible for the compiler to detect

the general cases of which these are particular examples, and hence to implement them efficiently.

5.1. Booleans

$$TYPE\ Bool \triangleq \forall T.T \rightarrow T \rightarrow T$$

In which $true \triangleq \lambda x.\lambda y.x$ and $false \triangleq \lambda x.\lambda y.y$. Any (terminating) function of this type must be equivalent to either *true* or *false*. The Ponder compiler takes advantage of this fact when generating code, so that any application of an object of that type to two arguments is generated as a test and jump, and so is as efficient as if *Bool* had been built in. In fact, the compiler recognises the general case of types of the form $\forall T.T \rightarrow T \rightarrow \ldots \rightarrow T$, and generates n-way branches as appropriate.

5.2. Pairs

$$TYPE\ Pair\ [A,\ B] \triangleq \forall T.(A \rightarrow B \rightarrow T) \rightarrow T$$

In which pairs are represented as functions which may be applied to $\lambda x.\lambda y.x$ or $\lambda x.\lambda y.y$ to return their first or second component respectively.

5.3. Options

Options are things which may or may not be there, like the tail of a list or the daughters of a node in a tree.

$$TYPE\ Option\ [T] \triangleq \forall R.(T \rightarrow R) \rightarrow R \rightarrow R$$

i.e. options are functions which either apply a function to the thing they hold, or return another result. In other words, if *opt* is an optional integer, then it will either be $\lambda f.\lambda x.f\ n$ for some integer n, or it will be $\lambda f.\lambda x.x$ (which is called *nil*), so

$$opt\ (\lambda x.2 \times x)\ 0$$

will wither return twice n if it is there, or zero if it isn't.

5.4. Lists

The previous two definitions allow us to define lists:

$$RECTYPE\ List\ [T] \triangleq Option\ [Pair\ [T,\ List\ [T]]]$$

Which means that a list will either be *nil* (as in Options above), or will accept a function to which it will pass the head and tail of the list.

5.5. Binary Unions

Unions are like options, except that the thing is either one type of thing or another type of thing:

$$TYPE\ Union\ [T_1,\ T_2] \triangleq \forall R.(T_1 \to R) \to (T_2 \to R) \to R$$

so if u is of type *Union* $[A, B]$ then $u\ f_a\ f_b$ applies either f_a if the object in the union is of type A, or f_b if it is of type B.

Note that in Milner's system abstract type definitions of *Bool*, *Pair*, *Option* and *Union* would not be allowed.

6. Inadequacies

Ponder has been used to write a large number of small programmes (an interactive calculator for example), and people using it feel confident enough to consider writing medium sized programmes (a spreadsheet system is under construction). Unfortunately a number of things are lacking and are difficult or impossible to provide without altering the language.

6.1. Unions

The most important example of an inadequacy in the type system is that it is not possible to define unions of the Algol 68 kind, in which the elements of the union are not ordered, so that one need only know the type of each alternative in order to select the appropriate cases, rather than having to remember the sequence of types within the union.

I can see no way of altering the type system to allow the definition of such unions, other than just adding them in in an *ad hoc* way (which I am loath to do). However, Ponder's unions are virtually the same as those in the current ML. Users of ML seem to have survived for quite some time with nothing more sophisticated, and the syntax-defining features of Ponder make their use somewhat more palatable.

6.2. Coercions

Another problem is that capsules are unrelated to all other types. For example if *Nat* is the same type as *Int* with restrictions to ensure that only positive integers are included, the type system would not allow one to pass a *Nat* quantity to a function requiring an *Int*, despite the fact that this would do no harm. Thus I am considering allowing the programmer to state that the relationship between the types in two capsules should be preserved.

Thus for example one could construct *Real* to be a superset of *Int*, allowing integers to be passed as arguments of functions of *Reals*. Unfortunately, the construction of related representations of related types can be quite tedious, and generally occurs in the wrong

order. One would be required to define all of *Int*, *Real* and *Complex* at once with this in mind, and to ensure that *Int*s were *Rational*s as well as *Real*s would be very difficult.

A possible solution to this would be to allow the programmer to specify that one type may be converted to another by the application of some arbitrary function (call it a coercion). This overcomes the objection stated above, but raises a whole new problem as to the order in which coercions should be applied. If we have coercion functions of type $Int \rightarrow Real$ and $Real \rightarrow Complex$, we can clearly use an *Int* when we want a *Complex*, but what if we also have $Int \rightarrow Rational$ and $Rational \rightarrow Complex$?

A further objection is that if the programmer is allowed to specify coercions, there is plenty of opportunity to make enormous mistakes. If we do not have them, though, the programme may have to be much larger. Consider the coercion $Int \rightarrow \forall T.Union[T, Int]$, which would save many injection functions.

Can we write a type checker which knows when to apply coercions, composes the appropriate conversion function, and is reasonably efficient? Consider passing a $Complex \rightarrow X$ to a $(Int \rightarrow X) \rightarrow Y$ function: the checker must compose the *Int* to *Real* and *Real* to *Complex* coercions with the argument in order to get an $Int \rightarrow X$ argument to give to the function. In [Mitchell 82], Mitchell describes an algorithm for doing this in Milner's system, but Ponder types are rather different, and overloading produces some other interesting problems.

7. Conclusion

The type checker described above has not been implemented. Instead, an earlier version is still in service. This previous version has several infelicities which cause it to reject some expressions which are type-valid. This was caused by an attempt to make the type-checker 'guess' a particular type for every sub-expression, something which although impossible to do accurately is desirable for helpful error messages.

Nonetheless, users of the Ponder compiler find the type-checking system sufficiently powerful for most of their requirements, and have been using the previous (pessimistic) type-checker for two years without much complaint.

8. Acknowledgements

I am grateful to Mike Gordon for patient supervision, and to Dave MacQueen for some useful conversations which helped to clarify my thoughts about the algorithms.

The work has been funded by the Science and Engineering Research Council of Great Britain.

Bibliography

[Church 41]: A. Church,
The Calculi of Lambda-Conversion,
Princeton University Press 1941

[Milner 78]: R. Milner,
A Theory of Type Polymorphism in Programming,
Journal of Computer and System Sciences Volume 17 No. 3, December 1978

[MacQueen 82]: D.B. MacQueen, Ravi Sethi,
A Semantic Model of Types for Applicative Languages,
Symposium on Lisp and Functional Programming 1982

[MacQueen 84]: D.B. MacQueen, Ravi Sethi,
An Ideal Model for Recursive Polymorphic Types,
Eleventh Annual ACM Symposium on Principles of Programming Languages 1984

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

[Mitchell 82]: J. C. Mitchell,
Coercion and Type Inference,
ACM Symposium on Lisp and Functional Programming 1982