Typechecking Polymorphism in Emerald

Andrew P. Black
Digital Equipment Corporation

Norman Hutchinson
University of Arizona

Digital Equipment Corporation
Cambridge Research Lab

CRL 91/1 (Revised) 8th July 1991

Abstract

Emerald is a statically typed object-oriented language that was originally intended for programming distributed subsystems and applications [Jul 88]. It is important that such systems be dynamically extensible, *i.e.*, that it be possible to introduce new kinds of entities into the system without re-compiling or re-linking the whole system. This led us to devise a type system based on the notion of type *conformity* rather than type equality. We also felt that polymorphism was a necessary feature of a modern programming language: programmers should be able to define generic abstractions like homogeneous sets, lists and files into which objects of arbitrary type can be placed. The combination of object-orientation, type-checking based on conformity and polymorphism gave rise to some interesting problems when designing Emerald's type system. This paper describes the Emerald's type-checking mechanism and the notion of types and parameterization on which it is based.

Keywords: object-oriented programming, static type checking, polymorphism, matching, F-bounded polymorphism, subtyping, conformity, extensibility, distributed programming.

Authors' electronic mail addresses: black@crl.dec.com; norm@cs.ubc.ca

This work was supported in part by the National Science Foundation under Grant CCR-8701516.

Norman Hutchinson was with the University of Arizona when this work was performed, and is now with the University of British Columbia.

1 Introduction

Emerald is an object-oriented language, by which we mean that the basic entities (objects) manipulated by programmers have identities as well as values. Emerald objects are said to be *autonomous*: each object "owns" its own set of operations, and these operations are the *only* ones that can be applied to the data that is used in the implementation of the object.

In a traditional value-oriented language, one of the major rôles of the type system is to prevent the misinterpretation of values, and thus to ensure that the meaning of a program is independent of the particular representation that is chosen for its data types [Donahue 85]. The rôle of types in an object-oriented language like Emerald is somewhat different, because autonomy and encapsulation already ensure that the only operations that can be applied to an object are those that belong to the object itself. Emerald's type system has other functions: earlier and more meaningful error detection and reporting, improved performance, and the classification of objects.

From the perspective of error detection, the assertion that an Emerald expression is type correct means that its evaluation will never lead to an object being requested to execute an operation that it does not possess. In Smalltalk terminology, typechecking guarantees that "operation not understood" errors are never generated. As a consequence, the runtime system need not check for such errors.

Before we proceed, we introduce some caveats. Emerald does not have subrange types of the kind found in Pascal; [0..9] and [5..19] are not types, and the question of whether either can be used in place of the other, or in the place of *integer*, does not arise. Neither are the bounds of the index of an array part of its type; as in CLU [Liskov 79], all Emerald arrays have flexible bounds. Emerald types never depend on ordinary non-type values; this is not because we do not believe that it is useful to specify the range of an integer variable, but because we regard the enforcement of such a specification as range checking, not type checking [Welsh 78].

Emerald also lacks an explicit notion of class. However, Emerald types induce a classification on Emerald objects: all objects that have a given type have an important property in common. Moreover, the conformity relation between types defines a lattice structure that is similar to a "multiple inheritance hierarchy". A group of objects that share the same implementation also have a common property, but our view, coloured by a strong belief in encapsulation and implementation experience on heterogeneous architectures, is that this particular property is of lesser interest to the programmer (although it is of great importance to the implementor.)

Having uttered these cautions as to what Emerald types are not, we are now ready to explain what they are. Speaking loosely, an Emerald type is a set of operation *names*: associated with each operation name is a signature that describes the arguments and results that the operation accepts and returns. Here is the type *Boolean* in Emerald:

$$\begin{array}{l}
\textbf{const}\ Boolean \leftarrow \\
\quad \textbf{typeobject}\ b \\
\qquad \textbf{operation}\ \wedge\ [b] \rightarrow [b] \\
\qquad \textbf{operation}\ \vee\ [b] \rightarrow [b] \\
\qquad \textbf{operation}\ \neg\ [\ ] \rightarrow [b] \\
\qquad \textbf{operation}\ =\ [b] \rightarrow [b] \\
\quad \textbf{end}\ b
\end{array}$$

This fragment of Emerald binds the name *Boolean* to the expression to the right of the $\leftarrow$; the fact that *Boolean* is declared **const** means that it cannot later be rebound. The expression **typeobject** b ... **end** b is a type constructor: it returns an Emerald object of type **type**. The identifier b is a bound variable that names this new object within the scope of the type constructor. The value of the type constructor represents a type with four operations, named $\wedge$, $\vee$, $\neg$ and $=$. Each of these operations takes one argument and one result, whose types are given by the expressions in the square brackets; recall that in an object-oriented language, every operation gets the target object as an implicit argument. Thus, if an object of type b receives the operation request $\wedge$, it expects a single argument of type b and produces a result of type b. (Although there is nothing in the type definition to tell us what this result might be, which we might reasonably expect it to be the conjunction of the values of the target and the argument.) An object-oriented version of $\neg$ would normally take no arguments, and would return the negation of the target. However, in this paper we will use a restricted version of Emerald in which *all* operations have a single argument and result; the empty argument list for $\neg$ formally represents an argument of type *Any*. This restriction is made to simplify the notation and exposition of the paper; dealing with the full generality of argument lists of unbounded size introduces significant notational complexity, but requires no new concepts.

Thus, we see that Emerald's view of types is quite different from that adopted by, for example, Russell: when Donahue and Demers [Donahue 85] state that types are set of operations, they mean that they are sets of *functions*; each function provides an interpretation of the underlying value space of representations. Emerald types are sets of *names*; the functions that are eventually applied to the representation are not a property of the type at all, but of the invoked object. As a consequence, Emerald types are *abstract*: it is possible for many different objects to be of the same type, even though they have different code bodies for their operations, and execute on different hardware architectures.

To illustrate this, here are two objects, both of which are *Boolean*, even though they have quite different implementations.

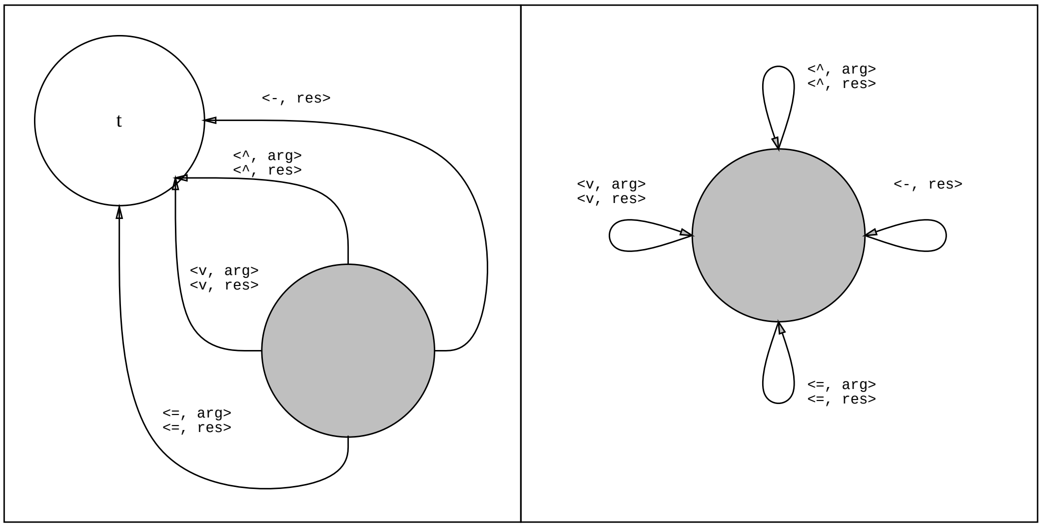
```
const true ←
  object t
    operation ∧ [a: Boolean] → [r: Boolean]
      r ← a
    operation ∨ [a: Boolean] → [r: Boolean]
      r ← t
    operation ¬ [ ] → [r: Boolean]
      r ← false
    operation = [a: Boolean] → [r: Boolean]
      r ← a
  end t
```

```
const false ←
  object f
    operation ∧ [a: Boolean] → [r: Boolean]
      r ← f
    operation ∨ [a: Boolean] → [r: Boolean]
      r ← a
    operation ¬ [ ] → [r: Boolean]
      r ← true
    operation = [a: Boolean] → [r: Boolean]
      r ← a.¬[ ]
  end t
```

Another important thing to notice about the definition of type *Boolean* is that it is self-referential; its local name b appears in its own body. This is typical of most types encountered in practice, and will turn out to be very important in our treatment of type checking. Formally, the meaning of such a definition is given as the fixedpoint of a function. Let $\lambda b.B(b)$ denote the function of the bound variable b corresponding to the Emerald type constructor **typeobject** $b \ldots$ **end** b; the description of exactly what this function is will be deferred to section 2, but for now it is sufficient to realize that it is a function from types to types, also known as a *type generator*. Intuitively, a *type* is a set of operator names and the corresponding signatures. The interesting properties of *Boolean* are captured by the type that is the fixedpoint of the generator $\lambda b.B(b)$, which we write as $\mathbf{Y}(\lambda b.B(b))$, or more concisely as $\mu b.B(b)$, following the notation introduced by MacQueen *et al.* [MacQueen 84, MacQueen 86].

Informally, we can represent this by the graph structures shown in Figure 1. The circles represent types; the shaded circles correspond to the captions. The arcs represent arguments and results of operations, as indicated by their labels. The graph on the left shows the type generator B applied to an arbitrary type argument t. The graph on the right represents the fixedpoint of B.

Unfortunately, there is one situation in which the notion of type as a mapping from operation names to signatures does not capture all of the information in an Emerald object constructor. As we will see in Section 4, dealing correctly with type parameters requires that we regard the constraint on a type parameteras a type *generator*, *i.e.*, as a function from types to types. This has little implication for the practising programmer,



B(t) μ t.B(t)

Figure 1: The type *Boolean* as a graph structure

since a type constructor such as **typeobject** *b* ... **end** *b* can be regarded equally well as representing a generating function or its fixedpoint.

The application areas for which Emerald is intended demand that systems be malleable and extensible. This means that it must be possible to introduce not just new objects, but new *types* of objects, into a running system, and that existing applications must be able to operate on these objects. In general, it is possible to substitute one object for another provided that the new object is at least as functional as the old one. This means that the new object must provide all of the operations offered by the old object, and that the arguments, results and semantics of these operations are appropriate. We regard the *syntactic* part of this substitutability condition to be the responsibility of the type system; ensuring that the *semantics* are appropriate involves program proving rather than type checking. A type system based upon equality of types will disallow many valid substitutions; Emerald's type system [Black 87] is based upon the notion of conformity, which is the largest relation between types that ensures that substitutions are safe. It can be argued that in some application areas there are reasons to use a more restricted rule. For example, in a centralized system in which all the types are introduced by one group of programmers, one might require an explicit statement of substitutability in addition to conformance; Trellis [Schaffert 86] is a language that takes this approach. However, using a rule that is less stringent than conformity will lead to an unsafe type system, *i.e.*, to a situation in which an undefined operation can be invoked on an object [Cook 89].

The remaining sections of this paper define conformity and explain how type checking is performed by the Emerald compiler. First we deal with the case where there are no type parameters; then we introduce them. For simplicity of presentation, we ignore two features of Emerald: the attributes that state whether an object is mutable and an operation is functional. As mentioned above, we also assume that all operations have a single argument and result, whereas in fact Emerald allows arbitrary numbers of arguments and results, and lets the same operator symbol be defined with different arities. This is handled formally by regarding the arity of a symbol to be part of its name. For example, nullary $+$ and unary $+$ are treated as the distinct function symbols $+_{0,1}$ and $+_{1,1}$, where the subscripts give the number of arguments and results.

The correct subscript for the operator in an invocation can always be inferred by counting the commas in the argument and result lists.

2 Types and Conformity

Emerald uses the notation $a \circ\!\!> b$ to mean that type a conforms to type b. The symbol $\circ\!\!>$ is intended to indicate that a has more operations than b, and that objects of type a can be used where those of type b are expected. (Some authors say that a is a *subtype* of b, but we will avoid this terminology; we find it confusing to call a type with *more* operations a *subtype*.)

Before defining conformity on types, we must state formally what a type is; this requires some subsidiary definitions. We regard a type as a mapping from operation names (elements of a denumerable set $\mathcal{F}$ defined by the language syntax) to *signatures*. Intuitively, the signature of an operation gives the types of its argument and result, and is represented as a pair of types, which we write $\langle a,\ r\rangle$. When we introduce type parameters, the types in a signature will no longer be constants, but may depend on the type or value of the argument; for this reason we define a signature to be a *function* from a type and a value to pairs of types. (However, until section 4, all these functions will be constant functions.) As mentioned in the introduction, in order to describe constraits on type parameters, we also need to deal with type *generators*, *i.e.*, functions from types to types. Thus, we have the following domains.

$$\begin{array}{lrcl} \text{Generators:} & \mathcal{G} & = & \mathcal{T} \rightarrow \mathcal{T} \\ \text{Types:} & \mathcal{T} & = & \mathcal{F} \rightarrow \mathcal{S} \\ \text{Signatures:} & \mathcal{S} & = & (\mathcal{T} \times \mathcal{V}) \rightarrow \mathcal{T} \times \mathcal{T} \end{array}$$

To make this more concrete, *Boolean* from Section 1 is the fixedpoint of the generator B given by

$$B = \lambda b.\lambda\phi.\begin{cases} s & \text{if } \phi = \wedge \\ s & \text{if } \phi = \vee \\ r & \text{if } \phi = \neg \\ s & \text{if } \phi = = \\ \mathit{wrong}_{\mathcal{S}} & \text{otherwise} \end{cases},$$

where $\mathit{wrong}_{\mathcal{S}}$ is a distinguished error signature, and where

$$\begin{array}{rcl} r & = & \lambda\langle t,\ v\rangle\,.\,\langle \mathit{Any},\ b\rangle \\ s & = & \lambda\langle t,\ v\rangle\,.\,\langle b,\ b\rangle\ . \end{array}$$

Usually, we will write functions like B more compactly as

$$B \quad = \quad \lambda b.\{\wedge \sim s, \vee \sim s, \neg \sim r, = \sim s\}\ .$$

Hence, the type *Boolean* that is the fixedpoint of this function is given by

$$\mathbf{Y}B \quad = \quad \mu b.\{\wedge \sim s, \vee \sim s, \neg \sim r, = \sim s\}\ .$$

Notation: Two types are of particular interest: *Any*, which corresponds to the type that has no operations at all, and *None*, which corresponds to the type with all possible operations, each having the most general possible signature.

$$\begin{array}{rcl} \mathit{Any} & = & \lambda\phi\,.\,\mathit{wrong}_{\mathcal{S}} \\ \mathit{None} & = & \lambda\phi\,.\,\lambda\langle t,\ v\rangle\,.\,\langle \mathit{Any},\ \mathit{None}\rangle \end{array}$$

For convenience, we define the following operations on types and signatures:

$$
\begin{array}{llll}
\mathsf{ops} & : \mathcal{T} \rightarrow 2^{\mathcal{F}} & \mathsf{ops}\ t & = \{\phi \in \mathcal{F} \mid t\phi \neq \mathit{wrong}_{\mathcal{S}}\} \\
\mathsf{sig} & : \mathcal{F} \rightarrow \mathcal{T} \rightarrow \mathcal{S} & \mathsf{sig}_{\phi} t & = \begin{cases} t\phi & \text{if } \phi \in \mathsf{ops}\ t \\ \lambda\langle g,\ v\rangle\,.\,\langle \mathit{None},\ \mathit{Any}\rangle & \text{otherwise} \end{cases} \\
\mathsf{arg} & : \mathcal{T} \times \mathcal{T} \rightarrow \mathcal{T} & \mathsf{arg}\ \langle t_1,\ t_2\rangle & = t_1 \\
\mathsf{res} & : \mathcal{T} \times \mathcal{T} \rightarrow \mathcal{T} & \mathsf{res}\ \langle t_1,\ t_2\rangle & = t_2
\end{array}
$$

Notation: When we define a relation $\mathcal{R}$ between types, we will also define a derived relation $\mathcal{R}_s$ for signatures.

$$s_1 \mathcal{R}_s s_2 \stackrel{\text{def}}{=} \forall\langle t,\ v\rangle\,.\,\mathsf{res}\ (s_1\langle t,\ v\rangle) \mathcal{R} \mathsf{res}\ (s_2\langle t,\ v\rangle)\ \wedge\ \mathsf{arg}\ (s_2\langle t,\ v\rangle) \mathcal{R} \mathsf{arg}\ (s_1\langle t,\ v\rangle)$$

For example, when we define conformity ($\circ\!\!>$) between types, we have also induced a relation $\circ\!\!>_{\mathcal{S}}$ on signatures: two signatures conform exactly when the result parts of those signatures conform, and the argument parts conform *inversely.*

$$s_1 \circ\!\!>_{\mathcal{S}} s_2 \stackrel{\text{def}}{=} \forall\langle t,\ v\rangle\,.\,\mathsf{res}\ (s_1\langle t,\ v\rangle) \circ\!\!> \mathsf{res}\ (s_2\langle t,\ v\rangle)\ \wedge\ \mathsf{arg}\ (s_2\langle t,\ v\rangle) \circ\!\!> \mathsf{arg}\ (s_1\langle t,\ v\rangle)$$

The contravariance of the arguments is quite natural, because the information flow associated with an argument is the inverse of that associated with a result. View a signature as a (partial) specification of an operation: one operation is stronger (more useful) than another if the result it delivers is stronger, or if the requirement that it places on its argument is weaker.

We formalize this notion of "more useful" using the idea that one type can be a *refinement* of another.

Definition (Refinement Relation): A relation $\mathcal{R}$ between types is said to be a *refinement* relation if and only if $t_1 \mathcal{R}\, t_2$ implies

1. $\mathsf{ops}\ t_1 \supseteq \mathsf{ops}\ t_2$
2. $\forall \phi \in \mathsf{ops}\ t_2 :\ \mathsf{sig}_{\phi} t_1\ \mathcal{R}_s\ \mathsf{sig}_{\phi} t_2$

Refinement represents the minimum requirement for substitutability; if an object of type t_1 is supplied where type t_2 is specified, then t_1 must possess all of t_2's operations, the results of each of t_1's operations must be usable where the result of the corresponding operation of t_2 was usable, and t_1's operation must be no more demanding in its arguments than was t_2's operation.

Now we are finally in a position to define Emerald's conformity relation.

Definition (conformity): Conformity (written $\circ\!\!>$) is the *maximal* refinement relation, *i.e.*, the one containing the most pairs.

The properties of conformity can be captured by inference rules; these rules represent theorems that can be proved from the definition of conformity, but we will not include the proofs here.

The first rule says that conformity is reflexive:

$$\frac{\Gamma \vdash a : \mathbf{type}}{\Gamma \vdash a \circ\!\!> a} \tag{1}$$

and should be read as follows: if one can prove in the naming environment Γ that the symbol a denotes a type, then one can infer that in the same environment, $a \circ\!\!> a$. We will usually omit antecedents like $a : \mathbf{type}$ in what follows.

The second rule covers the case where one type, f, has more operations than another type g; it lets us deduce that $f \circ\!\!> g$ whenever the signatures of the corresponding operations conform appropriately.

$$\frac{\begin{array}{c}\Gamma \vdash f = \mu t.\{\phi_1 \sim p_1, \ldots, \phi_n \sim p_n, \ldots, \phi_{n+m} \sim p_{n+m}\},\\ \Gamma \vdash g = \mu u.\{\phi_1 \sim q_1, \ldots, \phi_n \sim q_n\},\\ \Gamma, t = f, u = g \vdash \forall i \in [1, n].\, p_i \circ\!\!>_S q_i\end{array}}{\Gamma \vdash f \circ\!\!> g} \tag{2}$$

(Recall that types are mappings, and therefore that the ordering of the operation names between the braces is irrelevant.) This rule lets us create a conforming type both by adding operations to a type, and by generalizing the signatures of its existing operations. Note that since the expressions p_i may contain occurrences of t, and the q_i may contain u, it is necessary (in the third antecedent) to augment the environment Γ before proving that $p_i \circ\!\!>_S q_i$.

We introduce a special rule for *None*; because this type has an infinite number of operations it cannot be written as f in rule (2).

$$\frac{\Gamma \vdash a : \mathbf{type}}{\Gamma \vdash \mathit{None} \circ\!\!> a} \tag{3}$$

The final rule for conformity without type parameters is a generalization of rule (2) that deals with self referential and mutually referential types.

$$\frac{\begin{array}{c}\Gamma \vdash f = \mu t.\{\phi_1 \sim p_1, \ldots, \phi_n \sim p_n, \ldots, \phi_{n+m} \sim p_{n+m}\},\\ \Gamma \vdash g = \mu t.\{\phi_1 \sim q_1, \ldots, \phi_n \sim q_n\},\\ \Gamma, t = f, u = g, f \circ\!\!> g \vdash \forall i \in [1, n].\, p_i \circ\!\!>_S q_i\end{array}}{\Gamma \vdash f \circ\!\!> g} \tag{4}$$

This is the same as rule (2), except that in the third antecedent we may *assume* $f \circ\!\!> g$, the very proposition that we wish to prove. Intuitively, since f and g may refer not only to themselves but also to each other, in applying rule (2) we may have to show that $f \circ\!\!> g$. Rule (4) says that if we get into this situation, it is all right to assume what we are trying to prove, provided that we do not do so at the top level. Taking the converse point of view, the rule says that the only reason for two types not to conform is that *at some level* one does not have enough operations.

To see rule (4) in action, let us examine for conformity the three types A, E and I.

const $I \leftarrow$ **typeobject** *Int*
  operation $+$ [*Int*] $\rightarrow$ [*Int*]
  operation *isZero* [ ] $\rightarrow$ [*Boolean*]
  operation *neg* [ ] $\rightarrow$ [*Int*]
end *Int*

const $A \leftarrow$ **typeobject** *Addend*
  operation $+$ [I] $\rightarrow$ [*Addend*]
end *Addend*

const $E \leftarrow$ **typeobject** *Extra*
  operation $+$ [*Extra*] $\rightarrow$ [*Extra*]
end *Extra*

Rule (2) tells us that $I \circ\!\!> A$ if we can prove that, when $\mathit{Int} = I$ and $\mathit{Addend} = A$, the signature of $+$ in I $\circ\!\!>_S$ to the signature of $+$ in A. This requires that we prove that $I \circ\!\!> \mathit{Int}$ and $\mathit{Int} \circ\!\!> \mathit{Addend}$. The first condition follows from rule (1) but we have no way of proving the second condition. However, using rule (4) in place of rule (2), $\mathit{Int} \mathrel{\circ\!\!>} \mathit{Addend}$ follows from the extra assumption.

Rule (4) (correctly) does not permit us to prove $I \mathrel{\circ\!\!>} E$; even though I has more operations, the contravariance of signature conformity for the common operation, $+$, would require that $I \mathrel{\circ\!\!>} E$ *and* $E \mathrel{\circ\!\!>} I$. The latter condition is clearly false, and cannot be deduced from any of our rules.*

From our definition of conformity, we can derive an equivalence relation $\asymp$ by:

$$a \asymp b \stackrel{\text{def}}{=} a \mathrel{\circ\!\!>} b \wedge b \mathrel{\circ\!\!>} a$$

It can be shown that $\mathrel{\circ\!\!>}$ is a partial order over $\mathcal{T}_{\asymp}$, and *None* and *Any* are the maximum and minimum elements, respectively, under this ordering.

3 Type Checking

Emerald's fundamental theorem of typing guarantees that if a statement is type correct, then executing it will never cause an operation to be invoked on an object that does not possess that operation.

To state this theorem formally, we need to introduce the concepts of syntactic and dynamic type. Because Emerald objects are autonomous, it is possible to ask an object at runtime what operations it implements, and what their signatures are: from this one can deduce the *dynamic type* of the object. The Emerald expression **typeof** *exp* returns the dynamic type of the object that results from evaluating *exp*; the implementation of **typeof** requires, in general, dynamic interrogation of the target object.

In contrast, *syntactic type* is a property of an Emerald language expression, not an object. Syntactic type can always be determined statically. Emerald expressions are constructed recursively out of literals, identifiers, object constructors, and invocations. There is a corresponding set of rules that gives the syntactic type of the various forms of expression in terms of their constituent parts. If e is an expression and t is a type, we have so far written $e : t$ to mean that t is the syntactic type of e. It is sometimes convenient to have a function that returns the type of an expression e; we will write $\tau[\![e]\!]$ for this purpose. By definition, $e : \tau[\![e]\!]$.

Emerald *statements* do not have a type; they are either type-correct or erroneous. We will write $\surd[\![s]\!]$ to mean that the statement (or expression) s is type-correct. By convention, for an expression e, $\surd[\![e]\!] \equiv \tau[\![e]\!] \neq \mathit{wrong}_{\mathcal{T}}$. Here are the type-checking rules for the most important constructs in Emerald.

Declarations:

$$\frac{\tau[\![t]\!] = \textbf{type}}{\surd[\![\textbf{var}\ v : t]\!],\ \tau[\![v]\!] = t} \tag{5}$$

$$\frac{\tau[\![t]\!] = \textbf{type},\ \tau[\![e]\!] \mathrel{\circ\!\!>} t}{\surd[\![\textbf{const}\ c : t \leftarrow e]\!],\ \tau[\![c]\!] = t} \tag{6}$$

*Rule (4) is stronger (but less elegant) than the rule given by Cook *et al.*[Cook 90]:

$$\frac{\Gamma, r \mathrel{\circ\!\!>} s \vdash F(r) \mathrel{\circ\!\!>} G(s)}{\Gamma \vdash \mu t.F(t) \mathrel{\circ\!\!>} \mu t.G(t)}$$

This rule states that if the results of applying two type generators to conforming types also conform, then the fixedpoints of those generators conform. It deals with *self*-references in the type-generators F and G, but does *not* deal with references from F to G. For example, Cook's rule cannot be used to prove that $I \mathrel{\circ\!\!>} A$ above: when we compare the arguments of $+$ we are comparing a bound variable to a constant.

$$\frac{\surd[\![e]\!]}{\surd[\![\mathbf{const}\ c \leftarrow\ e]\!],\ \tau[\![c]\!] = \tau[\![e]\!]} \tag{7}$$

Expressions:

$$\frac{\surd[\![e]\!],\ \tau[\![t]\!] = \mathbf{type}}{\tau[\![\mathbf{view}\ e\ \mathbf{as}\ t]\!] = t} \tag{8}$$

$$\frac{\surd[\![a]\!],\ \phi \in \mathsf{ops}\ \tau[\![a]\!],\ \tau[\![e]\!]\ \circ\!\!>\ \mathsf{arg}\ (\mathsf{sig}_\phi \tau[\![a]\!]\ \langle \tau[\![e]\!],\ e\rangle)}{\tau[\![a.\phi[\,e\,]]\!] = \mathsf{res}\ (\mathsf{sig}_\phi \tau[\![a]\!]\langle \tau[\![e]\!],\ e\rangle)} \tag{9}$$

Statements:

$$\frac{\tau[\![e]\!]\ \circ\!\!>\ \tau[\![v]\!]}{\surd[\![v \leftarrow e]\!]} \tag{10}$$

Let us look briefly at rule (10): it states that for an assignment $v \leftarrow e$ to be type correct, the type of the expression e must conform to the type of the identifier that is being assigned. Rule (9) is the most interesting: it gives the typing conditions for the invocation $a.\phi[\,e\,]$. The principal antecedents are that ϕ be a defined operation on the syntactic type of a, and that the type of the argument conforms to the type declared for the argument of ϕ in a's type; if these conditions are satisfied, the type of the result is as given by a's type. Note that this rule specifies that the signature function $\mathsf{sig}_\phi \tau[\![a]\!]$ is applied to $\langle \tau[\![e]\!],\ e\rangle$. Since all of the signature functions that have been introduced so far in this paper have been constant functions, the choice of argument is not very interesting. We will return to this rule and explain the argument in Section 4.

Observe that all of the conditions for type correctness are phrased in terms of the syntactic type of the constituent elements. For $a.\phi[\,e\,]$ to be type correct it is not sufficient for the *object* bound to a to satisfy the given conditions; it must be possible to determine that the conditions are true of the *expression* a: this is what we mean by static type checking. If it turns out that the expression a does not have the required type, but the programmer knows through some other channel that the object bound to a will have a larger type, then a view expression can be used to create an expression of the required type, as shown in rule (8): the value of **view** e **as** t is the same as the value of e, but the type is always t. In general, execution of a view expression will require a run time check that the dynamic type of the object does indeed conform to the requested type.

In addition to the above, there are similar rules giving the types of literals, type constructors and object constructors, which are the only mechanisms whereby new objects may be introduced into the universe of an Emerald program. For each of these basic syntactic elements e, the following invariant holds:

$$\mathbf{typeof}\ e\ \ \circ\!\!>\ \ \tau[\![e]\!] \tag{11}$$

i.e., the dynamic type of the object that results from the evaluation of the literal conforms to the syntactic type of that literal[†]. By induction over the structure of the language, it can be shown that invariant (11) actually holds for *all* Emerald expressions. In particular, since an invocation can be shown to be type correct using only rule 9, it must be the case that the antecedent $\phi \in \mathsf{ops}\ \tau[\![a]\!]$ holds. This ensures our goal, *viz*, that in a type-correct program no operation will ever be invoked on an object that does not support it.

4 Type Parameters

The previous two sections have been concerned with the first of our requirements for Emerald's type system: that type checking be based on a conformity relation that is larger than equality. We now turn our attention

[†]In fact, for all the literals, **typeof** $e\ \mathrel{\underline{\overline{\circ}}}\ \tau[\![e]\!]$.

to the second requirement: that Emerald support statically typed polymorphism. Let us first consider an example of implicit polymorphism:

> **const** *map* $\leftarrow$ **typeobject** *m*
>   **operation** *apply*$[a : t] \rightarrow [r : t]$
>   **forall** *t*
> **end** *m*

Here is an object that has type *map*:

> **const** *identity* $\leftarrow$ **object** *id*
>   **operation** *apply*$[e : t] \rightarrow [r : t]$
>   **forall** *t*
> $r \leftarrow e$
>   **end** *apply*
> **end** *id*

The operation *apply* of *identity* simply returns its argument; it is the identity function. It is important that the syntactic type of the result be the same as that of the argument; this is achieved by use of the type parameter t, which is introduced by the clause **forall** t, and is bound to the syntactic type of the formal parameter a. The signature of *identity.apply* thus depends on the *type* of its argument e; we can now appreciate why signatures must be functions. Referring back to rule (9) in Section 3, we see that the argument to the signature function is $\langle\tau[\![e]\!],\ e\rangle$. The signature of *apply* in *map* is $\lambda\langle t,\ v\rangle.\langle t,\ t\rangle$; when applied to $\langle\tau[\![e]\!],\ e\rangle$ the result is $\langle\tau[\![e]\!],\ \tau[\![e]\!]\rangle$. In the invocation *identity.apply*$[e]$, the type of the argument e trivially conforms to $\mathsf{arg}\ \langle\tau[\![e]\!],\ \tau[\![e]\!]\rangle = \tau[\![e]\!]$, the conformity condition in the antecedent of rule (9) is always satisfied, and the invocation is type correct whenever the other conditions are satisfied. Moreover, the type of *identity.apply*$[e]$ is $\mathsf{res}\ \langle\tau[\![e]\!],\ \tau[\![e]\!]\rangle = \tau[\![e]\!]$ as required.

Let us now turn our attention to explaining the second element of the argument to the signatures in rule (9). This is e, by which we mean to convey the actual value of the expression e. (Strictly, we should introduce an evaluation function ϵ from expressions to values, but we avoid this as extraneous notation.) At first sight, passing a value to a signature function would appear to contradict our claim that typechecking in Emerald is concerned only with types, and never with values. To see why this is not in fact so, consider the following type, *emptySet*, which describes an object that creates empty sets of a specified type. (The notation $t \rhd eq$ and the rôle of *eq* will be explained in the next subsection.)

> **const** *emptySet* $\leftarrow$ **typeobject** *emptySet*
>   **operation** *of*$[t :$ **type**$] \rightarrow [r : x]$
>   **such that** $t \rhd eq$
>   **where** $eq \leftarrow$ **typeobject** *e*
>   **operation** $=[e] \rightarrow [Boolean]$
>   **end** *e*
>   **where** $x \leftarrow$ **typeobject** *set*
>   **operation** *insert*$[t] \rightarrow [\,]$
>   **operation** *extract*$[\,] \rightarrow [t]$
>   **end** *set*
> **end** *emptySet*

Suppose that an object *set* has type *emptySet*. We can then create a set of Booleans with:

> **const** *bset* $\leftarrow$ *set.of*$[Boolean]$

and a set of characters with:

```
const cset ← set.of[char]
```

Observe that the type of the result of an invocation of $set.of[t]$ must depend on the *value* of the argument t. Since t has type **type**, it is reasonable for the type system to be concerned with its value. In fact, the *only* values that are ever examined by the Emerald type system are **type** values.

We are now in a good position to attempt to write the signature of *of* on *emptySet*:

$$\lambda\langle t,\ v\rangle\,.\,\langle \mathbf{type},\ \mu x\,.\,\{insert \sim \lambda\langle t',\ v'\rangle\,.\,\langle v,\ Any\rangle,\ \ extract \sim \lambda\langle t',\ v'\rangle\,.\,\langle Any,\ v\rangle\}\rangle \tag{12}$$

As always, this signature is a function from a type and a value to a pair of types. The argument component is **type**, because the argument to *of* must have type **type**. The result component is a type in which the operation names *insert* and *extract* are mapped to constant signature functions, *i.e.*, *insert* and *extract* are not themselves polymorphic. However, the argument type of *insert* and the result type of *extract* do depend on v, the value of the argument given to *of*, as required.

Although it is promissing, the above signature does not address the two issues raised by the *emptySet* example: constrained types and the definition of conformity for type parameters.

Constrained Types

To guarantee that sets do not contain duplicate elements, we wish to require that the type argument t support an equality operation. This might be used by the implementation of *insert* to eliminate duplicates. The purpose of the **such that** clause is to express this requirement:

```
such that t ▷ eq
where eq ← typeobject e
   operation =[e] → [Boolean]
end e
```

The symbol ▷ (read matches) is used to introduce a constraint on a type parameter such as t. In earlier versions of Emerald, we used the symbol ⊳ to indicate such constraints. However, this is misleading, since the relation that we require between our type parameter and the constraint is not conformity. To see this, consider the following type.

```
const char ← typeobject char
   operation =[char] → [Boolean]
   operation ord[ ] → [int]
end char
```

Our definition of conformity (correctly) states that *char* ⋫ *eq*; if *char* did conform to *eq*, contravariance on the argument to = would require that *eq* conform to *char*. This cannot be the case because *eq* has fewer operations than *char*. We may think that this is an indication that our definition of conformity is incorrect, but this is not so. If we were to "correct" our definition of conformity in some way to allow *char* to conform to *eq*, then we would also allow *Boolean* (as defined in section 1) to conform to *eq*. If *char* ⊳ *eq* and *Boolean* ⊳ *eq* then the following program would be type correct.

```
var a, b : eq
a ← 'a'
b ← true
assert a.=[b]              % type error
```

To see why this cannot be allowed, notice that the assert statement will cause $a.=$ to be invoked with b as its argument. This is a type error, since $a.=$ requires a *char* as argument. (The implementation of $a.=$ might take advantage of the fact that its argument has type *char* by invoking *ord* on it; b does not possess this operation.)

So, our definition of conformity is correct in stating that *char* does not conform to *eq*. Nevertheless, *char* is a suitable argument for *emptySet.of*, since the result of that invocation is a homogeneous set in which the $=$ operation is applied only to objects of the same type as the target.

We have emphasized this point at such length for several reasons. First, although we first observed in reference [Hutchinson 87, Section 3.8] that conformity is not the appropriate relation to bound a type parameter, our overloading of the symbol $\circ\!\!>$ has probably led to confusion. Second, other authors have also used the same symbol for parameter constraints and for their "subtyping" relation. For example, America and van der Linden [America 90] first use $<$ to denote subtyping, which is essentially identical to our $\circ\!\!>$, but they then give an example in which an operation *sort* takes a type parameter written as "$X < Ordered$". In our notation, *Ordered* would be given by

typeobject *Ordered*
  operation $less[Ordered] \rightarrow [Boolean]$
end *Ordered*

and, using America's definition of subtyping, contravariance prevents *Int* and *Float* from being subtypes of *Ordered* in just the same way as (in our notation) $Int \not\circ\!\!> eq$. Later, America and van der Linden apply *sort* to both *Int* and *Float*; however, the interpretation of the symbol $<$ that permits this is never given.

The third reason for dealing with this topic in such detail is that the appropriate test to determine whether a type parameter satisfies a constraint, which we denote by $\triangleright$ (read *matches*), can be expressed only when constraints are modeled as type *generators* rather than as types. For a type $p : \mathcal{T}$ and a constraint $C : \mathcal{G}$

$$p \triangleright C \stackrel{\text{def}}{=} p \circ\!\!> C(p) \tag{13}$$

Matching appears to be very similar to *F-bounded polymorphism* as defined by Canning *et al.* [Canning 89]. Canning *et al.* define the F-bounding condition as $t \subseteq F[t]$, where $\subseteq$ is the subtyping realtion, and $F[t]$ is an *expression*, generally containing t. Because our approach is model-theoretic rather than proof-theoretic, it is natural for us to define matching in terms of a constraining *function* rather than an expression.

One consequence of this definition is that when we write an expresion like **such that** $t \triangleright eq$, in which a name like *eq* appears on the right hand side of the $\triangleright$ relation, that name must denote a function from types to types. The syntax of the programming language must therefore ensure that constraints can indeed be treated as functions. In our example, this has always been the case, since the constraint *eq* has been represented by a type constructor in which the self-referential structure is explicit. However, we have been less than rigorous in the discussion that opened this subsection and motivated the need for matching. Although we wrote $char \not\circ\!\!> eq$, if *eq* denotes a type generator, then in fact we showed that $char \not\circ\!\!> equal$, where $equal = \mathbf{Y}eq$ is a type constant.

const $equal \leftarrow$ **typeobject** e
  operation $=[e] \rightarrow [Boolean]$
end e

In practice, it is always possible to infer the need to take the fixedpoint of a type generator, so no confusion need arise. However, in the remainder of this paper, we will take care to use *eq* to mean the type generator, and *equal* to mean its fixedpoint.

The relation $\triangleright$ captures the similarity in self-referential structure between two types. In particular, although $char \not\triangleright equal$, $char \triangleright eq$, as is readily seen by substituting *char* for the formal parameter e in the expression that defines the function *eq*.

We can now see what is missing from the signature for *emptySet.of* that we wrote in (12): it does not capture the constraint on the type argument of *of*. We extend (12) by adding this condition:

$$\lambda\langle t,\ v\rangle\,.\quad \begin{array}{ll} \textbf{if} & v \triangleright eq \\ \textbf{then} & \langle \textbf{type},\ \mu x\,.\,\{insert \sim \lambda\langle t',\ v'\rangle\,.\,\langle v,\ Any\rangle,\ \ extract \sim \lambda\langle t',\ v'\rangle\,.\,\langle Any,\ v\rangle\}\rangle \\ \textbf{else} & \langle wrong_{\mathcal{T}},\ wrong_{\mathcal{T}}\rangle \end{array} \tag{14}$$

Conformity of type parameters

When we type-check an Emerald program that does not contain type parameters, all of the types are manifest. This is necessary for the term static type checking to be meaningful, and is enforced by the syntax of the language. However, when we type-check the body of an operation with a type parameter, the values that the parameter may take on, although constant throughout the scope of the operation, are no longer manifest. Indeed, it is implicit in the concept of statically checking an operation with a type parameter that the operation body is deemed type-correct only if *all* valid parameterizations of that body are correct.

To illustrate this, consider the following operation body.

```
const o ← object o
   operation illegal [a : t] → [r : Boolean]
      forall t
      r ← a.¬[ ]
   end illegal
end o
```

If *o.illegal* is applied to *true*, t is bound to *Boolean* and the body gives rise to no type error. Nevertheless, the body is not type-correct as written because the type of o also allows *o.illegal* to be applied to a character. To make it o correct, we must add an appropriate constraint on t, *e.g.*, by changing the quantifier to

```
forall t such that t ▷ not
where not ← typeobject n
   operation ¬ [ ] → [Boolean]
end n .
```

Thus it is clear that the constraint on a type parameter must play a central rôle in the type-checking process. In the model, a type parameter is treated as the set of types that match the constraint; any statement involving the type parameter $\pi\ :\ \textbf{par} \triangleright C$ is true only if it is true for every member of the set $\{t \mid t \triangleright C\}$.

Most of the existing model and inference rules, which were developed with constant types in mind, are easily adapted to type parameters. For example, if π is a type parameter constrained by C (which we will write as $\pi\ :\ \textbf{par} \triangleright C$), then ops $\pi \supseteq$ ops $\mathbf{Y}C$. However, we do need new rules for the conformance properties of type parameters.

The obvious candidate rule,

$$\frac{\Gamma \vdash \pi\ :\ \textbf{par} \triangleright C\ \wedge\ \mathbf{Y}C \circ\!\!> a}{\Gamma \vdash \pi \circ\!\!> a}$$

in which conformity of a type parameter follows from the conformity of the fixedpoint of its constraint, turns out not to hold. To see this, consider the object returned by an invocation of *emptySet.of*. This object might well contain the following declaration of an array to hold the representation of the set:

var *Rep*: $Array.of[t]$,

where t : **par** $\triangleright\, eq$. The *extract* operation might use the expression

$Rep.removeLower[\,]$,

the type of which is t. To what identifiers can this expression be legally assigned? Clearly, the following code must be permitted:

operation $extract[\,] \rightarrow [result : t]$
  $result \leftarrow Rep.removeLower[\,]$
end *extract* .

In other words, when t : **par** $\triangleright\, eq$, $t \;\circ\!\!> t$.

However, the following is *incorrect* according to our model:

operation $badExtract[\,] \rightarrow [result : equal]$
  $result \leftarrow Rep.removeLower[\,]$
end *badExtract* .

We have already seen that *char* $\triangleright t$, so the set used to model t contains *char*. Thus, the assigment in *badExtract* can be type correct only if *char* $\circ\!\!>$ *equal*, and we have already shown that this is not so. However, our candidate inference rule would let us deduce that $t \;\circ\!\!>$ *equal*, because $\mathbf{Y}eq \;\circ\!\!>$ *equal* (indeed, $\mathbf{Y}eq =$ *equal*), and so it must be rejected.

It turns out that the correct rule is

$$\frac{\Gamma \vdash \pi \;:\; \mathbf{par} \triangleright C \;\wedge\; C(\pi) \circ\!\!> a}{\Gamma \vdash \pi \circ\!\!> a} \quad ; \tag{15}$$

if the constraint (considered as a type generator) applied to π conforms to a, then we can deduce that $\pi \circ\!\!> a$.

Although it is difficult to provide the intuition behind this rule, it is easily justified by the model.

Theorem: Inference rule (15) is sound with respect to the model.

Proof (sketch): The type parameter π is modeled by the set of types matching the constraint C:

$$\begin{aligned} \pi &= \{t \mid t \triangleright C\} \\ &= \{t \mid t \circ\!\!> C(t)\} \quad \text{(by definition of } \triangleright) \end{aligned}$$

Hence,

$$\forall t \in \pi \;.\; t \circ\!\!> C(t) \quad . \tag{16}$$

The second conjunct in the antecedent, $C(\pi) \circ\!\!> a$, gives us, by the definition of conformity for sets:

$$\forall t \in \pi \;.\; C(t) \circ\!\!> a \quad . \tag{17}$$

Combining (16) and (17), we get

$$\forall t \in \pi \;.\; t \circ\!\!> C(t) \wedge C(t) \circ\!\!> a \tag{18}$$

from which we can deduce, by transitivity of $\circ\!\!>$,

$$\forall t \in \pi \;.\; t \circ\!\!> a \tag{19}$$

which is equivalent to $\pi \mathrel{\circ\!\!>} a$, the required consequent. [‡]

So far we have examined the conformance of a type parameter to other types. Now let us look at conformity in the reverse direction. If π is a type parameter constrained by C, then the model tells us that $a \mathrel{\circ\!\!>} \pi$ only if $a = None$ or $a = \pi$. This is because π contains all the types that are "stronger" than C, *i.e.*, they may have many more operations. However, since these conformities can be deduced from rules (1) and (3), no new inference rules are needed.

Because the antecedent of rule (15) requires us to prove $C(\pi) \mathrel{\circ\!\!>} a$, in attempting to apply this rule we may become involved in proving that $\pi \mathrel{\circ\!\!>} a'$, or that $a'' \mathrel{\circ\!\!>} \pi$, for some types a' and a''. The latter can be derived by use of rules (1) and (3); the former requires the recursive application of rule (15).

We can now state a fundamental result: once the body of an operation that contains a type parameter has been type checked using the rules given above, the body will also be type correct if the type parameter is replaced by any type that matches the constraint.

Theorem: Given an operation body $B(\pi)$, where π is a type parameter constrained by C, then if we can prove that $B(\pi)$ is type correct using the rules given in this paper, then it follows that $B(t)$ is also type correct for any type t that matches C. In symbols:

$$\frac{\Gamma \vdash \pi \,:\, \mathbf{par} \rhd C \;\wedge\; \sqrt{[\![B(\pi)]\!]}}{\Gamma, t \rhd C \vdash \sqrt{[\![B(t)]\!]}}$$

This result follows immediately from the soundness of the inference rules with respect to a model in which a type parameter is treated as representing the set of all matching types.

5 Summary

This paper has described the type system of the Emerald programming language. The basic notion underlying the type system, conformity, has been described previously with varying degrees of formality [Schaffert 86, Black 87, Cook 90, America 90]. Our contributions are the extension of the basic concepts to allow both implicit and explicit polymorphism, which require the recognition that the bounding constraints of parameters cannot be expressed using conformity alone, but need the notion of matching, or F-boundedness. Matching cannot be expressed if constraints are treated as mappings from operation names to signatures; it is necessary to recognize that a constraint is a generating function whose fixedpoint is such a mapping. This is reminiscent of Cook's recognition that inheritance requires that classes be treated as the fixedpoints of generating functions, and that it is these generators, and not the classes themselves, that are inherited.

It is important to recognize that, although the notion of conformity is very powerful, it is by itself insufficient to describe constrained types. The combination of matching with the rule for the conformity of type parameters results in our being able to type-check the body of a parametric procedure once when it is declared, rather than whenever it is parameterized. Thus, operations with type parameters become first class citizens in Emerald.

[‡] It is tempting to strengthen rule (15) further, in the same way that rule (2) was strengthened to become rule (4). It would then become

$$\frac{\begin{array}{c}\Gamma \vdash \pi \,:\, \mathbf{par} \rhd C,\\ \Gamma, \pi \mathrel{\circ\!\!>} a \vdash C(\pi) \mathrel{\circ\!\!>} a\end{array}}{\Gamma \vdash \pi \mathrel{\circ\!\!>} a} \quad . \tag{20}$$

However, we have not been able to prove this rule sound (or unsound); neither have we been able to find a example in which the extra assumption is necessary.

6 Acknowledgements

The work reported in this paper has taken place over a period of years. The authors wish to thank Martín Abadi, Laurence Carter, William Cook, Alan Demers, Albert Meyer, Vaughan Pratt, Jon Riecke and Mark Tuttle for many helpful and stimulating discussions. We also thank Luca Cardelli for encouraging us to strive for ever-higher standards of rigor in our presentation.

References

[America 90] Pierre America and Frank van der Linden. A parallel object-oriented language with inheritance and subtyping. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 161–168. ACM, October 1990. Published in *SIGPLAN Notices*, 25(10), October 1990.

[Black 87] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65–76, January 1987.

[Canning 89] Peter S. Canning, William R. Cook, Walter L. Hill, Walter Olthoff, and John C. Mitchell. F-Bounded polymorphism for object-oriented programming. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[Cook 89] W. R. Cook. A proposal for making Eiffel type-safe. *Computer Journal*, 32(4):305–311, August 1989.

[Cook 90] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, January 1990.

[Donahue 85] James Donahue and Alan Demers. Data types are values. *ACM Transactions on Programming Languages and Systems*, 7(3):426–445, July 1985.

[Hutchinson 87] Norman Hutchinson. *Emerald: An Object-Oriented Language for Distributed Programming.* PhD thesis, Department of Computer Science, University of Washington, January 1987.

[Jul 88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[Liskov 79] Barbara Liskov, Russ Atkinson, Toby Bloom, Eliot Moss, Craig Schaffert, Bob Scheifler, and Alan Snyder. CLU reference manual. Technical Report TR-225, Massachusetts Institute of Technolgy, Laboratory for Computer Science, October 1979.

[MacQueen 84] David MacQueen, Gordon Plotkin, and Ravi Sethi. An Ideal model for recursive polymorphic types. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 165–174, January 1984.

[MacQueen 86] David MacQueen, Gordon Plotkin, and Ravi Sethi. An Ideal model for recursive polymorphic types. *Information and Control*, 71(1):95–130, October/November 1986.

[Schaffert 86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl. In *Proc. First Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 9–16, Portland, OR, October 1986. ACM.

[Welsh 78] J. Welsh. Economic range checks in Pascal. *Software — Practice and Experience*, 8:85–97, 1978.