Dismissing the "Final Concern"

or

Matches Rides Again

A Position Paper for the ANSA workshop on F-bounded quantification

Andrew P. Black
Digital Equipment Corporation Cambridge Research Laboratory

6th July 1992

1 Introduction

In his paper "Revising the DPL Type System" [Watson 92], Andrew Watson concludes by raising a "final concern" about the suitability of the matches relation $\triangleright$ for constraining the type of a parameter to a polymorphic operation. It is the purpose of this note to lay this "final concern" to rest, and to dispel any lingering doubts as to the suitability of $\triangleright$ for expressing parameter constraints.

2 Matches and F-bounded Polymorphism

The F-bounding condition is defined by Canning *et al.* [Canning 89] as $t \subseteq F[t]$, where $\subseteq$ is the subtyping relation and $F[t]$ is an *expression*, generally containing the type variable t. Because Canning's work is proof theoretic, this essentially syntactic definition is natural.

In our technical report "Typechecking Polymorphism in Emerald", Hutchinson and I define the relation $\triangleright$ (read matches) used to constrain type parameters in Emerald [Black 91]. Here the setting is model-theoretic: if t is a type, and C is a type generator (a function from types to types), then

$$p \triangleright C \stackrel{\text{def}}{=} p \mathrel{\circ\!\!>} C(p) \tag{1}$$

where $\circ\!\!>$ is Emerald's subtyping relation (called conformity).

Notation

In the remainder of this position paper the notation of the Emerald Programming Language will be used. In addition, rather than depending explicitly on the types-as-functions model developed in reference [Black 91], the notation

$$\mathsf{type}\{\theta[a] \rightarrow [r],\ \psi[b] \rightarrow [s]\}$$

will be used to denote a type, without regard for how that type is modeled. Objects with the above type possess the two operations named θ and ψ; θ has signature $[a]\rightarrow[r]$, which means that it takes an argument of type a and returns a result of type r, while ψ takes an argument of type b and returns a result of type s. Further, λ and μ will be used in the usual way, so that

$$G = \lambda t.\ \mathsf{type}\{\phi[a] \rightarrow [t]\}$$

Author's electronic mail addresses: black@crl.dec.com

is a function that maps types into types, and

$$f = \mu t.\ \mathsf{type}\big\{\phi[a] \rightarrow [t]\big\}$$

is the fixpoint of that function. Thus $f = G(f) = \mathsf{type}\big\{\phi[a] \rightarrow [f]\big\}$; this provides a way of writing self-referential types.

3 Parametric Polymorphism and Type Constraints

The purpose of this section is to motivate (by the use of an example) the rôle of matching (F-bounding) in describing parameter constraints. This section can be omitted by those familiar with the need for matching.

One of the simplest motivating examples is an object that constructs homogeneous sets, *i.e.*, an object with an operation *of* that takes as argument a type t, and returns an empty set into which objects of type t can be inserted. The type of such a constructor might be declared as follows.

```
const emptySet ← typeobject emptySet
   operation of[t : type] → [r : x]
      suchthat t ▷ eq
      where eq ← typegenerator e
         operation =[e] → [Boolean]
      end e
      where x ← typeobject set
         operation insert[t] → [ ]
         operation extract[ ] → [t]
      end set
end emptySet
```

In the implementation of the *insert* operation, the argument must be tested for equality with elements already in the set. The **suchthat** clause expresses the constraint that the type argument to *of* possess such an equality operation.

The type *char* is a suitable element type for a set, and should be a legal argument to the *of* operation.

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

Notice that *eq* is defined as a type generator, whereas *char* is a type.

$$\begin{aligned} eq &= \lambda e.\ \mathsf{type}\big\{=[e] \rightarrow [Boolean]\big\} \\ char &= \mu c.\ \mathsf{type}\big\{=[c] \rightarrow [Boolean],\ ord[\,] \rightarrow [int]\big\} \end{aligned}$$

Observe that $char \not\mathrel{\circ\!\!>} \mathsf{fix}\,eq$, because $\mathsf{fix}\,eq = \mu e.\ \mathsf{type}\big\{=[e] \rightarrow [Boolean]\big\}$, and if *char* were to conform to this type, contravariance on the argument to $=$ would require that $\mathsf{fix}\,eq$ conform to *char*. This cannot be the case because $\mathsf{fix}\,eq$ does not have the *ord* operation of *char*.

Now consider $eq(char) = \mathsf{type}\big\{=[char] \rightarrow [Boolean]\big\}$. Clearly $char \mathrel{\circ\!\!>} eq(char)$. Hence, by the definition of $\triangleright$ (1), we have $char \triangleright eq$. Thus the use of $\triangleright$ to constrain the parameter to *of* allows *of* to be applied to *char*, whereas the use of $\mathrel{\circ\!\!>}$ does not.

4 Watson's Concern

Watson points out that the function $\lambda t.\ t \rhd C$ for some fixed C is not monotone in t [Watson 92, Section 5.4.2.5]. In other words,

$$h \mathrel{\circ\!\!>} i \ \wedge\ i \rhd C \ \not\Rightarrow\ h \rhd C\ .$$

To see this, consider the example

$$\begin{aligned} h &= \mathsf{type}\{\alpha[\,] \to [i],\ \beta[\,] \to [\,],\ \gamma[\,] \to [\,]\} \\ i &= \mu t.\ \mathsf{type}\{\alpha[\,] \to [t],\ \beta[\,] \to [\,]\} \\ C &= \lambda t.\ \mathsf{type}\{\alpha[\,] \to [t]\} \end{aligned}$$

Clearly, $h \mathrel{\circ\!\!>} i$, and $i \mathrel{\circ\!\!>} C(i)$, so $i \rhd C$. But $h \not\mathrel{\circ\!\!>} C(h)$ because the result of α in h is i, while the result of α in $C(h)$ is h, and $i \not\mathrel{\circ\!\!>} h$. Hence $h \not\rhd C$.

Why might this be a concern? Imagine that $o.\theta$ has signature $[T]\to[T]$ forall $T \rhd C$. Further, suppose that e is an expression with syntactic type i. (For example, e might be a name declared as **var** e: i.) The semantic function $\mathcal{T}$ is used to obtain the syntactic type of an expression, so in this case we have $\mathcal{T}[\![e]\!] = i$. Then $o.\theta[e]$ is type-correct, since $\mathcal{T}[\![e]\!] \rhd C$.

However, when e is actually evaluated, it might well yield an object with type h; since $h \mathrel{\circ\!\!>} i$, this is permitted by our type checking regime. Nevertheless, since $h \not\rhd C$, the invocation $o.\theta[$**view** e **as** $h]$ is type incorrect. †

Watson's concern arises because both $o.\theta[e]$ and $o.\theta[$**view** e **as** $h]$ must in fact invoke the same operations on the same objects. Since $o.\theta[e]$ is type-correct we know that these invocations cannot cause a "message not understood" error. Surely, then, $o.\theta[$**view** e **as** $h]$ must also be considered to be type correct?

5 An Important Omission

What is omitted from this reasoning is that the type-checking rule for invocations does not merely tell us that $o.\theta[e]$ is type-correct, but also assigns it a syntactic type [Black 91, rule 9]. Given that $o.\theta$ has signature $[T]\to[T]$ forall $T \rhd C$, we have

$$\mathcal{T}[\![o.\theta[e]]\!] = \mathcal{T}[\![e]\!]\ . \tag{2}$$

Now consider a possible implementation of operation θ.

operation θ $[a\!:\!T] \to [r\!:\!T]$ **forall** T **suchthat** $T \rhd C$
  $r \leftarrow a.\alpha[\,]$
end θ

The constraint on the type of a gives us enough information to guarantee not just that a has an α operation, but also that the result of $a.\alpha[\,]$ has syntactic type T, and thus that the assignment to r is type-correct. The semantics of this implementation do therefore satisfy the type-checking rule (2).

Now consider once again the invocation $o.\theta[$**view** e **as** $h]$, and further assume that the object yielded by the evaluation of e does in fact have dynamic type h. In the body of θ, the invocation $a.\alpha[\,]$ will still be

†The Emerald expression **view** e **as** h evaluates to the same object as the expression e, but has syntactic type h. In the usual case that $\mathcal{T}[\![e]\!] \not\mathrel{\circ\!\!>} h$, the evaluation of the view expression will require a conformance check at run time.

understood, but the result will have syntactic type i, not h. The result of $o.\theta[\textbf{view } e \textbf{ as } h]$ will therefore also have type i; this violates rule (2), which in this case becomes

$$\mathcal{T}[\![o.\theta[\textbf{view } e \textbf{ as } h]]\!] = \mathcal{T}[\![\textbf{view } e \textbf{ as } h]\!] = h$$

Now we see why $o.\theta[\textbf{view } e \textbf{ as } h]$ must be considered to be a type error. It is not because there might be an interaction error in the body of the θ operation: as Watson rightly summises, this cannot occur. It is because allowing this invocation would erroneously cause us to assign the syntactic type h to its result, when in fact we ought to assign it type i. An interaction error could then occur in the calling code, when operation γ is invoked on the result of $o.\theta[\textbf{view } e \textbf{ as } h]$.

6 A Simpler Example

Assuming that the reader is convinced by the argument so far, the obvious question that arises is what would happen if the operation whose argument is constrained by $\triangleright$ in fact returns no result at all, or returns a result whose type is independent of that of its argument.

To examine this situation, imagine that object o possesses an additional operation ψ with signature $[T]\rightarrow[Any]$ forall $T \triangleright C$. Corresponding to this signature is the implementation

operation ψ $[a\colon T] \rightarrow [r\colon Any]$ **forall** T **suchthat** $T \triangleright C$
  $r \leftarrow a.\alpha[\,]$
end ψ .

Just as with θ, $o.\psi[e]$ is type correct, but $o.\psi[\textbf{view } e \textbf{ as } h]$ is type incorrect. However, since (the results of) all invocations of $o.\psi$ have syntactic type Any, and since Any possesses no operations, no subsequent interaction error can occur. It seems that in this case it is overly pessimistic to make $o.\psi[\textbf{view } e \textbf{ as } h]$ a type error.

The reader should note carefully that the responsibility for this pessimism lies not with the type checking rules, nor with the definition of $\triangleright$, but with the programmer who selected the constraint for ψ. If the signature for ψ were instead $[T]\rightarrow[Any]$ forall $T \triangleright D$, where

$$D = \lambda t.\ \mathsf{type}\big\{\alpha[\,] \rightarrow [Any]\big\} \quad ,$$

and if the implementation of ψ is revised to read

operation ψ $[a\colon T] \rightarrow [r\colon Any]$ **forall** T **suchthat** $T \triangleright D$
  $r \leftarrow a.\alpha[\,]$
end ψ ,

then the body of ψ would still be type-correct. However, since $i \triangleright D$ and $h \triangleright D$, both the invocations $o.\psi[e]$ and $o.\psi[\textbf{view } e \textbf{ as } h]$ are type-correct.

Why might a programmer use C rather than D to constrain the argument to ψ? One reason might be a simple mistake, or a misunderstanding of the meaning of type constraints. Another reason might be that he or she intends to allow for other implementations of ψ (on other objects), for which the more stringent constraint might be necessary. In either case, the decision as to whether $o.\psi[\textbf{view } e \textbf{ as } h]$ should be type-correct is in the hands of the programmer.

7 Summary

This paper has shown that the $\triangleright$ relation used to constrain parameters of polymorphic operations in Emerald is robust to the situation described by Watson. When a strong constraint is necessary in order to maintain the type-checking invariant, $\triangleright$ enables the programmer to express that constraint. Of course, it is possible for a programmer to write a constraint that is more stringent than necessary for the type-correctness of a particular piece of code; this is analogous to the situation that arises with monomorphic operations, where it is possible for the programmer to require that the arguments possess operations that are never used.

References

[Black 91] Andrew P. Black and Norman Hutchinson. Typechecking polymorphism in Emerald. Technical Report CRL TR 91/1 (Revised), Digital Equipment Corporation, Cambridge Research Laboratory, July 1991. Available by electronic mail from techreports@crl.dec.com.

[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.

[Watson 92] Andrew J. Watson. Revising the DPL type system. Technical Report APM/RC.339.02, Architecture Projects Management Limited, June 1992.