Epigram: Practical Programming with Dependent Types

Conor McBride

School of Computer Science and Information Technology,
University of Nottingham, Jubilee Campus,
Wollaton Road, Nottingham NG8 1BB, United Kingdom
ctm@cs.nott.ac.uk

1 Motivation

Find the type error in the following Haskell expression:

```
if null xs then tail xs else xs
```

You can't, of course: this program is obviously nonsense unless you're a type-checker. The trouble is that only certain computations make sense if the `null xs` test is `True`, whilst others make sense if it is `False`. However, as far as the type system is concerned, the type of the `then` branch is the type of the `else` branch is the type of the entire conditional. Statically, the test is irrelevant. Which is odd, because if the test really were irrelevant, we wouldn't do it. Of course, `tail []` doesn't go wrong—well-typed programs don't go wrong—so we'd better pick a different word for the way they do go.

Abstraction and application, tupling and projection: these provide the 'software engineering' superstructure for programs, and our familiar type systems ensure that these operations are used compatibly. However, sooner or later, most programs inspect data and make a choice—at that point our familiar type systems fall silent. They simply can't talk about specific data. All this time, we thought our *programming* was strongly typed, when it was just our *software engineering*. In order to do better, we need a static language capable of expressing the significance of particular values in legitimizing some computations rather than others. We should not give up on programming.

James McKinna and I designed Epigram [27,26] to support a way of programming which builds more of the intended meaning of functions and data into their types. Its *style* draws heavily from the Alf system [13,21]; its *substance* from my to Randy Pollack's Lego system [20,23] Epigram is in its infancy and its implementation is somewhat primitive. We certainly haven't got everything right, nor have we yet implemented the whole design. We hope we've got *something* right. In these notes, I hope to demonstrate that such nonsense as we have seen above is not inevitable in real life, and that the extra articulacy which dependent types offer is both useful and *usable*. In doing so, I seek to stretch your imaginations towards what programming can be if we choose to make it so.

V. Vene and T. Uustalu (Eds.): AFP 2004, LNCS 3622, pp. 130–170, 2005.

1.1 What Are Dependent Types?

Dependent type systems, invented by Per Martin-Löf [22] generalize the usual function types $S \to T$ to dependent function types $\forall x : S \Rightarrow T$, where T may mention—hence *depend* on—x. We still write $S \to T$ when T doesn't depend on x. For example, matrix multiplication may be typed[1]

$$\mathbf{mult} \; : \; \forall i, j, k : \mathsf{Nat} \Rightarrow \mathbf{Matrix}\; i\; j \; \to \; \mathbf{Matrix}\; j\; k \; \to \; \mathbf{Matrix}\; i\; k$$

Datatypes like $\mathbf{Matrix}\; i\; j$ may depend on values fixing some particular property of their elements—a natural number indicating size is but one example. A function can specialize its return type to suit each argument. The typing rules for abstraction and application show how:

$$\frac{x : S \vdash t \; : \; T}{\lambda x \Rightarrow t \; : \; \forall x : S \Rightarrow T} \qquad \frac{f \; : \; \forall x : S \Rightarrow T \quad s \; : \; S}{f \; s \; : \; [s/x]T}$$

Correspondingly, $\mathbf{mult}\; 2\; 3\; 1 \; : \; \mathbf{Matrix}\; 2\; 3 \; \to \; \mathbf{Matrix}\; 3\; 1 \; \to \; \mathbf{Matrix}\; 2\; 1$ is the specialized multiplier for matrices of the given sizes.

We're used to universal quantification expressing *polymorphism*, but the quantification is usually over types. Now we can quantify over all values, and these include the types, which are values in $\star$. Our $\forall$ captures many forms of abstraction uniformly. We can also see $\forall x : S \Rightarrow T$ as a logical formula and its inhabitants as a function which computes a *proof* of $[s/x]T$ given a particular value s in S. It's this correspondence between programs and proofs, the *Curry-Howard Isomorphism*, with the slogan 'Propositions-as-Types', which makes dependent type systems particularly suitable for representing computational logics.

However, if you want to check dependent types, be careful! Look again at the application rule; watch s hopping over the copula,[2] from the term side in the argument hypothesis (eg., our specific dimensions) to the type side in the conclusion (eg., our specific matrix types). With expressions in types, we must think again about when types are equal. Good old syntactic equality won't do: $\mathbf{mult}\; (1{+}1)$ should have the same type as $\mathbf{mult}\; 2$, so $\mathbf{Matrix}\; (1{+}1)\; 1$ should *be* the same type as $\mathbf{Matrix}\; 2\; 1$! If we want computation to preserve types, we need at least to identify types with the same *normal forms*. Typechecking requires the *evaluation* of previously typechecked expressions—the phase distinction is still there, but it's slipperier.

What I like about dependent types is their precise language of data structures. In Haskell, we could define a sequence of types for lists of fixed lengths

```
data List0 x = Nil
data List1 x = Cons0 x (List0 x)
data List2 x = Cons1 x (List1 x)
```

[1] We may write $\forall x : X; y : Y \Rightarrow T$ for $\forall x : X \Rightarrow \forall y : Y \Rightarrow T$ and $\forall x_1, x_2 : X \Rightarrow T$ for $\forall x_1 : X; x_2 : X \Rightarrow T$. We may drop the type annotation where inferrable.

[2] By *copula*, I mean the ':' which in these notes is used to link a term to its typing: Haskell uses '::', and the bold use the set-theoretic '$\in$'.