Recursive type equivalence for datatypes

June 3, 2021

The motivation for wanting to extend the notion of type equivalence for recursive types in the MIL is the failure of the current system to deal with two issues that arise in checking datatype specs against their implementation.

Firstly, mutually recursive datatypes that are declared in a different order in the implementation from the interface will not match.

Secondly, a mutually recursive datatype will not match against a spec where at least one (but not all) of the arms are held abstract.

A basic motivating example we can work with is as follows:

```
local
  functor F(type d1
            datatype d3 = E of d2
                 and d2 = C of d3 | D of int * d1) = ....

  structure S =
   struct
     datatype d1 = A | B of d1*d2
          and d2 = C of d3 | D of int * d1
          and d3 = E of d2
   end

in
  structure T = F S
end
```

In our current system, the type portion of this code would look something like:

$$\begin{array}{rcl} F_c & = & \Lambda(x : \{d_1 :: T, d :: S(\mu(\alpha,\beta).(\beta, \alpha + int \times d_1\})), d_3 :: S(d.1), d_2 :: S(d.2)).\ldots \\ F_e & = & \Lambda(x : \{d_1 :: T, d :: S(\mu(\alpha,\beta).(\beta, \alpha + int \times d_1\})), d_3 :: S(d.1), d_2 :: S(d.2)). \\ & & \lambda(x : \{E : d_2 \to d_3, C : d_3 \to d_2, D : int \times d_1 \to d_2, \text{expose}_{d_3} : d_3 \to d_2, \text{expose}_{d_2} : d_2 \to d_3 \end{array}$$

$$\begin{array}{rcl} S_c & = & \text{Let } d = \mu(\alpha,\beta,\gamma).(1+\alpha\times\beta, \gamma + int\times\alpha, \beta) \text{ in } \{d = d, d_1 = d.1, d_2 = d.2, d_3 = d.3\} \\ S_e & = & \ldots \\ T_c & = & F_c(S_c) \\ T_e & = & F_e(S_e) \end{array}$$

Unfortunately, the application $F_c(S_c)$ is not well typed given our current notion of type equivalence.

Possible solutions to this include

1. Strengthening the notion of type equivalence to allow the application to be well typed.

2. Changing how we compile datatypes so that the types will correspond.

The solution is constrained by the meta-level constraint that we wish to be able to inline constructors: that is, we should be able to predict the implementation of the term level constructor and destructor functions for the datatypes.

1 Strengthening equivalence (Shao's equation)

Full Amadio/Cardelli recursive type equivalence [**?**] solves the problem, but is more than we need. Full AC equivalence says that two types are equal if their infinite unrollings are equal. This introduces two kinds of equations that we do not need:

$$\Gamma \vdash \mu(\alpha).int \equiv int \tag{1}$$

$$\Gamma \vdash \mu(\alpha).int\times\alpha \equiv \mu(\alpha).int\times(int\times\alpha) \tag{2}$$

Equation **??** is undesireable since it is likely that float and $\mu(\alpha)$.float would have different representations at the machine level. Similarly, equation **??** seems to require that pairs and recursives types have the same implementation. (This is probably less problematic than the former?)

A weaker notion of equivalence is to say that two *recursive* types are equal if their infinite unrollings are equal. This eliminates equation **??**, but still allows equation **??**

Shao's equation is a still weaker form of equivalence that seems to correspond to the idea that two recursive types are equal if their infinite in-place expansions are equal. This can be thought of as AC where we leave the μ binding sites intact as markers when we "unroll". Neither of the above equations are deriveable in this system.

Using the Abadi/Cardelli formalization, this corresponds to adding the following two inference rules:

$$\frac{}{\Gamma \vdash \mu(\alpha).F[\alpha] \equiv \mu(\alpha).F[\mu(\alpha).F[\alpha]]} \tag{3}$$

$$\frac{\Gamma \vdash \mu(\alpha).c1 \equiv F[\mu(\alpha).c1] \qquad \Gamma \vdash \mu(\alpha).c2 \equiv F[\mu(\alpha).c2] \qquad F[t]\downarrow t}{\Gamma \vdash \mu(\alpha).c1 \equiv \mu(\alpha).c2} \tag{4}$$

1.1 Implementation issues

It seems likely that this can be implemented with just small modification to the normal AC algorithm. Unfortunately this would need to be done at both the HIL and MIL levels. This should not interact with type inference since datatypes should not contain meta-variables.

Given these extended equivalence rules, it seems that we could continue to compile datatypes as we normally do. The inlined constructors are no problem, since we can predict the implementation, and the types of the constructors will work out to be equivalent to what we predicted.

2 Coercion interpretation