Hi Grigore,

I have a feeling that we in fact disagree on nothing.

Let me use a simple IMP program as an example to illustrate what I mean.

Remind that we have Maude module called IMP that builds all patterns.

```
fmod IMP is
  sorts Variable Trm Trm? Predicate Pattern ... .
  ...
  op ite : Pattern{Bool} Pattern{Pgm} Pattern{Pgm}
        -> Pattern{Pgm} [frozen(2 3)] .
  op while : Pattern{Bool} Pattern{Pgm}
          -> Pattern{Pgm} [frozen(1 2)] .
  ---- dereferencing a pointer
  op dereference : Pattern{Nat} -> Pattern{Nat} .
  ...
endfm
```

My objective is to write a Maude module called EXE that reduces configurations of IMP programs according to their semantics.

```
fmod EXE is
  vars P Q : Pattern{Pgm} . var B : Pattern{Bool} .
  eq ite(tt, P, Q) = P .
  eq ite(ff, P, Q) = Q .
  eq while(B, P) = ite(B, seq(P, while(B, P)), skip) .
  ...
endfm

reduce

#cfg(#k(seq(while(...), ...)),
     #state(merge(binder(three, one), ...))))

.
```

Notice how "frozen" attributes play a role here.

The look-up rule is the problem.

```
   #cfg(C[dereference(X)], C'[binder(X, V)])
=> #cfg(C[V], C'[binder(X, V)]) .
```

This look-up rule has two context variables C and C'. It is easy to instantiate C' as follows

```
   #cfg(C[dereference(X)], #state(merge(binder(X, V), H)))
=> #cfg(C[V], #state(merge(binder(X, V), H))) .
```

where variable H can match the rest of the heap, thanks to the associativity and commutativity of merge and that emp being the identity of merge.

The context C, though, has an infinite number of instances. For example,

```
   #cfg(#k(asgn(Y, dereference(X))),
        #state(merge(binder(X, V), H)))
=> #cfg(#k(asgn(Y, V)),
        #state(merge(binder(X, V), H)))

   #cfg(#k(asgn(Y, succ(dereference(X)))),
        #state(merge(binder(X, V), H)))
=> #cfg(#k(asgn(Y, V)),
        #state(merge(binder(X, V), H)))

   #cfg(#k(asgn(Y, succ(succ(dereference(X))))),
        #state(merge(binder(X, V), H)))
=> #cfg(#k(asgn(Y, V)),
        #state(merge(binder(X, V), H)))

   #cfg(#k(asgn(Y, succ(succ(succ(dereference(X)))))),
        #state(merge(binder(X, V), H)))
=> #cfg(#k(asgn(Y, V)),
        #state(merge(binder(X, V), H)))

        ......
```

Of course, given an initial configuration, we can write a Maude module `EXE` with a finite number of rules that reduces *that* configuration all the way to the end. My argument is that we cannot write a Maude module which can reduce *all* configurations, because that will need an infinite number of rules, unless we explicitly deal with context splitting, plugging, refocusing, etc.

I have been reading Traian's "*A rewriting logic approach to operational semantics*" and I found many good references in that paper. I have been looking into reduction semantics and techniques (such as refocusing) that helps to implement an efficient interpreter. I will keep reading literatures for a while.

I also intend to implement an algorithm in Maude that calculates the equivalence class under context application of a pattern φ, which is the set of all pairs of context and redex (C, R) such that $\varphi = C[R]$.

Yours,
Xiaohong