An effect system for dynamic instances

Albert ten Napel

1 Problem description

In order for our effect system to be practical and precise, it should fulfill the following requirements:

- A function with no observable effects (encapsulated effects) has no effects in its type.
- A function that has no effects in its type has no observable effects.
- An effect handler should be able to remove effects if they are completely handled.
- Instance creation that is not observational should not appear in the type.
- Instance creation that is observational should be reflected in the type.

It is important that encapsulated effects will not appear in the type, since these effects are not observable outside of the function.

Consider the following example.

```
effect Flip {
  flip : () -> Bool
}

f () =
  inst <- new Flip;
  inst#flip ()
```

Here we define a `Flip` effect which has a single operation `flip` that takes a unit value and returns a boolean.

The function `g` creates an new instance of `Flip` and calls `flip` on it. This operation call remains unhandled and so `g` is not a pure function. It is even impossible to handle this effect since the instance `inst` is needed, but this instance never leaves the scope of `g`. This means that the effect of creating `inst` also is observational and should be reflected in the type.

Next consider the following function `g`.

```
g () =
  inst <- new Flip;
  with handle {
    inst#flip () k -> k True
  } handle inst#flip ()
```

The function `f` created a new instance of the `Flip` effect, calls `flip` on the instance and immediately handles this operation call by returning True. Although `f` creates instances and call operations, these are not observable effects, since the operation call is immediately handled by the in-line handler and the instance does not escape its scope. Because these effects are not observable we would like `f` to be given the pure type `() -> Bool`. In fact `f` is functionally the same as the constant function that always returns `True`.

Consider the following function `h`.

```
h () =
  inst <- new Flip;
  ()
```

Although `h` creates a new instance, this instance is never used or returned and so the effect of its creation is not observed outside of the function. As such we would like this function have the pure type of `() -> ()`.

Consider the following function `h'`.

```
h' () =
  inst <- new Flip;
  inst
```

`h'` both creates a new instance and returns it. Although `h'` does not call any operations, its creation of the instance is an observational effect and so this should be reflected in its type.

Consider the following function `h''`.

```
h'' () =
  i1 <- h' ();
  i2 <- h' ();
  (i1, i2)
```

Here we are calling the previous function `h'` twice and so we are also creating two distinct instances. In the type it should be reflected that we are creating two distinct instances.

Consider the following function `u`.

```
u b =
  if b then
    i1 <- new Flip;
    i1
  else
    i2 <- new Flip;
    i2
```

`u` creates a new instance in each branch of the if expression. The observational effect of creating a new instance of the `Flip` is exactly the same whether `b` is `True` or `False`, so one would expect the type of `u` to only mention one new instance. To keep our effect system simpler we will however show two new instances in the type. Because these instances are distinct the return type will be an instance type with two instances instead on one.

Consider the following function `u'`.

```
u' b =
  if b then
    i1 <- new Flip;
    (i1, i1)
  else
```

```
i2 <- new Flip;
i3 <- new Flip;
(i2, i3)
```

Here `u'` creates a different amount of instances in each branch of the if expression. All three instances will occur in the type. In order to match the types of the branches the two pairs have to be matched. This will result in two instance types where the first contains `i1` and `i2` and the second `i1` and `i3`.

TODO: how to type instances as arguments? (polymorphic instances)
TODO: problems that come from general recursion, dynamic amount of instances

2 Effect system

We extend computation types to also contain the instances that are created. Computation types are now of the form: $I@t!E$, where I is a set of instance names, t is a value type and E is a set of pairs of instances with operations of the form $i\#o$. A valid computation type does not contain instance names that are not in the set I.

Using our proposed system we give the above functions the following types:
`f` : $() \to \{inst\}@\texttt{Bool}!\{inst\#\texttt{flip}\}$
`g` : $() \to \texttt{Bool}$
`h` : $() \to ()$
`h'` : $() \to \{inst\}@\texttt{Flip}^{\{inst\}}!\{\}$
`h''` : $() \to \{a, b\}@(\texttt{Flip}^{\{a\}}, \texttt{Flip}^{\{b\}})!\{\}$
`u` : $Bool \to \{i1, i2\}@\texttt{Flip}^{\{i1,i2\}}!\{\}$
`u'` : $Bool \to \{i1, i2, 3\}@(\texttt{Flip}^{\{i1,i2\}}, \texttt{Flip}^{\{i1,i3\}})!\{\}$

For typing the creation of new instances we lift the instance name to the type level and typecheck c with the instance added to the environment. There are two cases to consider now, either the instance is actually observationally used and will appear type of c or the instance is not observation-

ally used and will not appear in the type of c. If the instance is observationally used, we add it to the set of created instances. If the instance is not observationally used, we do not add it. This makes sure created instances will only appear in the type if they are actually observationally used.

$$\frac{\Gamma, x : E^{\{x\}} \vdash c : I@T \quad x \in T}{\Gamma \vdash x \leftarrow \texttt{new}\ E; c : (\{x\} \cup I)@T} \qquad \frac{\Gamma, x : E^{\{x\}} \vdash c : I@T \quad x \notin T}{\Gamma \vdash x \leftarrow \texttt{new}\ E; c : I@T}$$

For computation sequencing the two computation have to have the same type and effect annotations and we simply compute the union of the two created instance sets.

$$\frac{\Gamma \vdash c_1 : I_1@\tau!E \quad \Gamma, x : \tau \vdash c_2 : I_2@\tau!E}{\Gamma \vdash (x \leftarrow c_1; c_2) : (I_1 \cup I_2)@\tau!E}$$

The interesting part is function application. Here it is possible to create two distinct instances with the same name. In order make sure that we still have to distinct names in the created instance set we have to create fresh names for all the instances in the function v_1 and substitute these for the names in the return type.

$$\frac{\Gamma \vdash v_1 : \tau \to I@T \quad \Gamma \vdash v_2 : \tau \quad I' := \text{fresh names for each name in } I}{\Gamma \vdash v_1\ v_2 : I'@[I'/I]T}$$