An effect system for dynamic instances

Albert ten Napel

1 Effect handler effect system

1.1 Introduction

We extend a simplified version of the type system described by Bauer and Pretnar in [1]. We simplify the system from that paper by removing static instances and always having handlers contain all operations from a single effect. We extend the system with instance type variables, existential computation types and a computation to dynamically create instances. For the effect annotations on the computation types (called the dirt in [1]) we will take sets of instance variables.

1.2 Syntax

We assume there is set of effect names $E = \{\varepsilon_1, ..., \varepsilon_n\}$. Each effect has a set of operation names $O_\varepsilon = \{op_1, ..., op_n\}$. We every operation name only corresponds to a single effect. Each operation has a parameter type τ_{op}^1 and a return type τ_{op}^2. Annotations r are sets of instance variables.

$$
\begin{array}{lll}
\tau ::= & & \text{(value types)} \\
\quad i, j, k & & \text{(instance variables)} \\
\quad () & & \text{(unit type)} \\
\quad \tau \rightarrow \underline{\tau} & & \text{(type of functions)} \\
\quad \underline{\tau} \Rightarrow \underline{\tau} & & \text{(type of handlers)} \\
\underline{\tau} ::= & & \text{(computation types)} \\
\quad \tau \, ! \, r & & \text{(annotated type)} \\
\quad \exists(i : \varepsilon).\underline{\tau} & & \text{(existential)} \\
\nu ::= & & \text{(values)} \\
\quad x, y, z, k & & \text{(variables)} \\
\quad () & & \text{(unit value)} \\
\quad \lambda x.c & & \text{(abstraction)} \\
\quad handler(\nu) \, \{return \, x \rightarrow c, op_1(x; k) \rightarrow c, ..., op_n(x; k) \rightarrow c\} & & \text{(handler)} \\
c ::= & & \text{(computations)} \\
\quad return \, \nu & & \text{(return value as computation)} \\
\quad \nu \, \nu & & \text{(application)} \\
\quad x \leftarrow c; c & & \text{(sequencing)} \\
\quad with \, \nu \, handle \, c & & \text{(handler application)} \\
\quad \nu \# op(\nu; y.c) & & \text{(operation call)} \\
\quad new \, \varepsilon & & \text{(instance creation)}
\end{array}
$$

1.3 Subtyping rules

The subtyping rules are mostly the same as the rules described in [1], but we add two rules for the instance variables and the existential types. These are as you would expect.

$$\overline{() <: ()}$$

$$\frac{\begin{array}{c} a' <: a \\ b <: b' \end{array}}{a \to b <: a' \to b'}$$

$$\frac{\begin{array}{c} a' <: a \\ b <: b' \end{array}}{a \Rightarrow b <: a' \Rightarrow b'}$$

$$\frac{\begin{array}{c} a <: a' \\ e \subseteq e' \end{array}}{a \,!\, e <: a' \,!\, e'}$$

$$\overline{i <: i}$$

$$\frac{a <: b}{\exists(i : \varepsilon).a <: \exists(i : \varepsilon).b}$$

1.4 Well-formedness judgement

We have a well-formedness judgement for both value and computation types $\Delta \vdash \tau$ and $\Delta \vdash \underline{\tau}$. Where Δ stores bindings of instance variables to effects. For instance variables we simply check that they occur in Δ. For computation types we check that all the variables in the annotation occur in Δ.

$$\overline{\Delta \vdash ()}$$

$$\frac{\Delta[i] = \varepsilon}{\Delta \vdash i}$$

$$\frac{\begin{array}{c} \Delta \vdash a \\ \Delta \vdash b \end{array}}{\Delta \vdash a \to b}$$

$$\frac{\begin{array}{c} \Delta \vdash a \\ \Delta \vdash b \end{array}}{\Delta \vdash a \Rightarrow b}$$

$$\frac{\begin{array}{c} \Delta \vdash a \\ \Delta \vdash j_i \end{array}}{\Delta \vdash a \,!\, \{j_0, ..., j_n\}}$$

$$\frac{\Delta, i : \varepsilon \vdash a}{\Delta \vdash \exists(i : \varepsilon).a}$$

3

1.5 Typing rules

For the typing rules there are two judgments, $\Delta; \Gamma \vdash \nu : \tau$ for assigning types to values and $\Delta; \Gamma \vdash c : \underline{\tau}$ for assigning computation types to computations. Γ stores bindings of variables to types and Δ stores bindings of instance variables to effects ε. The function FIV returns the set of free instances of a type and the function IV returns the set of all instance variable of a type, bound or not.

In the sub-typing and abstraction rules we check that the introduced types are well-formed, so that we don't introduce ill-formed types in the context.

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash \nu : \tau_1 \\ \Delta \vdash \tau_2 \\ \tau_1 <: \tau_2 \end{array}}{\Delta; \Gamma \vdash \nu : \tau_2} \qquad \frac{\Gamma[x] = \tau}{\Delta; \Gamma \vdash x : \tau} \qquad \frac{}{\Delta; \Gamma \vdash () : ()} \qquad \frac{\begin{array}{c} \Delta \vdash \tau_1 \\ \Delta; \Gamma, x : \tau_1 \vdash c : \underline{\tau_2} \end{array}}{\Delta; \Gamma \vdash \lambda x.c : \tau_1 \to \underline{\tau_2}}$$

In the following rule
$h = handler(\nu) \{return\ x_r \to c_r, op_1(x_1; k_1) \to c_1, ..., op_n(x_n; k_n) \to c_n\}$.
For the handlers we first check that value ν is bound to an instance variable i of effect ε. We check that the operations in the handlers are exactly the operations belonging to ε. After we check that all the cases in the handler agree on the return type and annotations. We have the condition $r_1 \backslash \{i\} \subseteq r_2$ to make sure that any extra effects before the handler remain unhandled after the handler is evaluated. We check that the types on the left and right sides of the handler type are well-formed to make sure that no instance variables are introduced that are not in the context Δ.

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash \nu : i \\ \Delta[i] = \varepsilon \\ O_\varepsilon = \{op_1, ..., op_n\} \\ \Delta; \Gamma, x_r : \tau_1 \vdash c_r : \tau_2\ !\ r_2 \\ \Delta; \Gamma, x_i : \tau_{op_i}^1, k_i : \tau_{op_i}^2 \to \tau_2\ !\ r_2 \vdash c_i : \tau_2\ !\ r_2 \\ r_1 \backslash \{i\} \subseteq r_2 \\ \Delta \vdash \tau_1\ !\ r_1 \\ \Delta \vdash \tau_2\ !\ r_2 \end{array}}{\Delta; \Gamma \vdash h : \tau_1\ !\ r_1 \Rightarrow \tau_2\ !\ r_2}$$

The following rules are straightforward and similar to the corresponding rules in [1].

$$\frac{\Delta;\Gamma \vdash c : \underline{\tau}_1 \quad \Delta \vdash \underline{\tau}_2 \quad \underline{\tau}_1 <: \underline{\tau}_2}{\Delta;\Gamma \vdash c : \underline{\tau}_2} \qquad \frac{\Delta;\Gamma \vdash \nu : \tau}{\Delta;\Gamma \vdash return\ \nu : \tau\ !\ \varnothing}$$

$$\frac{\Delta;\Gamma \vdash \nu_1 : \tau_1 \to \underline{\tau}_2 \quad \Delta;\Gamma \vdash \nu_2 : \tau_1}{\Delta;\Gamma \vdash \nu_1\ \nu_2 : \underline{\tau}_2} \qquad \frac{\Delta;\Gamma \vdash \nu : \underline{\tau}_1 \Rightarrow \underline{\tau}_2 \quad \Delta;\Gamma \vdash c : \underline{\tau}_1}{\Delta;\Gamma \vdash with\ \nu\ handle\ c : \underline{\tau}_2}$$

For the operation calls we check that the value ν_1 is bound to an instance variable i of effect ε. We check that the operation belongs to ε and that the value ν_2 is of the parameter type of the operation. We then typecheck the continuation c and make sure that the instance variable i is in the annotation on the type of c.

For the creation of instances we return an existential type to account for the newly created instance.

$$\frac{\begin{array}{l}\Delta;\Gamma \vdash \nu_1 : i \\ \Delta[i] = \varepsilon \\ op \in O_\varepsilon \\ \Delta;\Gamma \vdash \nu_2 : \tau_{op}^1 \\ \Delta;\Gamma, y : \tau_{op}^2 \vdash c : \tau\ !\ e \\ i \in e\end{array}}{\Delta;\Gamma \vdash \nu_1\#op(\nu_2; y.c) : \tau\ !\ e} \qquad \frac{}{\Delta;\Gamma \vdash new\ \varepsilon : \exists(i : \varepsilon).i\ !\ \varnothing}$$

We can introduce and remove existential quantifiers at any point so long as the instance variable is not contained in the free instance variable of the type. With the third rule we can also rename instance variables if the name is not contained in the type.

$$\frac{\Delta;\Gamma \vdash c : \underline{\tau} \quad i \notin FIV(\underline{\tau})}{\Delta;\Gamma \vdash c : \exists(i : \varepsilon).\underline{\tau}} \qquad \frac{\Delta;\Gamma \vdash c : \exists(i : \varepsilon).\underline{\tau} \quad i \notin FIV(\underline{\tau})}{\Delta;\Gamma \vdash c : \underline{\tau}} \qquad \frac{\Delta;\Gamma \vdash c : \exists(i : \varepsilon).\underline{\tau} \quad j \notin IV(\underline{\tau})}{\Delta;\Gamma \vdash c : \exists(j : \varepsilon).\underline{\tau}[i := j]}$$

For sequencing we want both c_1 and c_2 to have existential types, where $\bar{i}$ and $\bar{j}$ are sequenced of bindings. If either c_1 or c_2 do not have existential types we can introduce them with the previous rule. To typecheck c_2 with all the bindings $\bar{i}$ introduced in the context Δ. For the return type we concatenate all the bindings $\bar{i}$ and $\bar{j}$ and we take the union of the annotations r_1 of c_1 and the annotations r_2 of c_2.

$$\frac{\Delta; \Gamma \vdash c_1 : \exists \bar{i}.\tau_1 \ ! \ r_1 \qquad \Delta, \bar{i}; \Gamma, x : \tau_1 \vdash c_2 : \exists \bar{j}.\tau_2 \ ! \ r_2}{\Delta; \Gamma \vdash x \leftarrow c_1; c_2 : \exists \bar{i}.\exists \bar{j}.\tau_2 \ ! \ (r_1 \cup r_2)}$$

1.6 Examples

We will show some examples together with the types that the discussed type system will assign to them. For the examples we will assume that the following effect is in the context.

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

The following function f creates a new instances and calls an operation on it, but does not return the instance itself. In the type we have an existential but we see that the instance does not appear in the value type, but only in the effect annotation. This way we know that we do not have access to the instance and so are unable to handle the effect.

```
f : () -> exists (i:Flip). Bool!{i}
f () =
  inst <- new Flip;
  inst#flip ()
```

In the following function g we create a new instance, call an operation on it and then immediately handle this effect. The type of this function is pure, since the effect is immediately handled.

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

The following function is also pure because although an instance is created, this instance is never used or returned.

```
g : () -> ()!{}
g () =
  inst <- new Flip;
  ()
```

The following function simply wraps the creation of a *Flip* instance.

```
h' : () -> exists (i:Flip). i!{}
h' () =
  inst <- new Flip;
  inst
```

In h'' we use h' twice. This means that we create two distinct *Flip* instances and we also expect two instance variables bound in the type.

```
h'' : () -> exists (i:Flip) (j:Flip). (i, j)!{}
h'' () =
  i1 <- h' ();
  i2 <- h' ();
  (i1, i2)
```

References

[1] Bauer, Andrej, and Matija Pretnar. "An effect system for algebraic effects and handlers." International Conference on Algebra and Coalgebra in Computer Science. Springer, Berlin, Heidelberg, 2013.