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

The syntax stays the same as in [1], but we add dynamic instance creation and instance values. Instance values represent a certain index at run-time, indexed by an $n \in \mathbb{N}$. They are used to define the semantics of the system but would not appear in an user-facing language. The only way to get an instance value is by using the *new* ε computation. We assume there is set of effect names $E = \{\varepsilon_1, ..., \varepsilon_n\}$. Each effect has a set of operation names $O_\varepsilon = \{op_1, ..., op_n\}$. Every operation name only corresponds to a single effect. Each operation has a parameter type τ^1_{op} and a return type τ^2_{op}. Annotations r are sets of instance variables.

$\tau ::=$	(value types)
$\quad i, j, k$	(instance variables)
$\quad ()$	(unit type)
$\quad \tau \to \underline{\tau}$	(type of functions)
$\quad \underline{\tau} \Rightarrow \underline{\tau}$	(type of handlers)
$\underline{\tau} ::=$	(computation types)
$\quad \tau \,!\, r$	(annotated type)
$\quad \exists(i : \varepsilon).\underline{\tau}$	(existential)
$\nu ::=$	(values)
$\quad x, y, z, k$	(variables)
$\quad ()$	(unit value)
$\quad \mathsf{inst}(n)$	(instance values)
$\quad \lambda x.c$	(abstraction)
$\quad \mathit{handler}(\nu)\ \{\mathit{return}\ x \to c, \mathit{op}_1(x; k) \to c, ..., \mathit{op}_n(x; k) \to c\}$	(handler)
$c ::=$	(computations)
$\quad \mathit{return}\ \nu$	(return value as computation)
$\quad \nu\ \nu$	(application)
$\quad x \leftarrow c; c$	(sequencing)
$\quad \mathit{with}\ \nu\ \mathit{handle}\ c$	(handler application)
$\quad \nu \# \mathit{op}(\nu; y.c)$	(operation call)
$\quad \mathit{new}\ \varepsilon$	(instance creation)

1.3 Subtyping rules

The subtyping rules are mostly the same as the rules described in [1]. These are as you would expect.

$$\frac{}{() <: ()} \qquad \frac{a' <: a \quad b <: b'}{a \to b <: a' \to b'}$$

$$\frac{a' <: a \quad b <: b'}{a \Rightarrow b <: a' \Rightarrow b'} \qquad \frac{a <: a' \quad e \subseteq e'}{a \,!\, e <: a' \,!\, e'}$$

We add rules for instance variables and existentials, which we compare structurally.

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

For existentials we are allowed to remove one if the instance variable does not appear in the type (where $FIV(a)$ is the set of free instance variables of a). We are also allowed to swap two existentials.

$$\frac{a <: b \quad i \notin FIV(a)}{\exists(i : \varepsilon).a <: b} \qquad \frac{}{\exists(i : \varepsilon).\exists(j : \varepsilon').a <: \exists(j : \varepsilon').\exists(i : \varepsilon).a}$$

We can prove reflexivity and transitivity of the subtyping relation from these rules.

Theorem 1 (Subtyping reflexivity)**.** *for all value and computation types a, $a <: a$*

Theorem 2 (Subtyping transitivity)**.** *for all value and computation types a, b and c, if $a <: b$ and $b <: c$ then $a <: c$*

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.

$$\frac{}{\Delta \vdash ()} \qquad \frac{(i : \varepsilon) \in \Delta}{\Delta \vdash i} \qquad \frac{\Delta \vdash a \quad \Delta \vdash b}{\Delta \vdash a \to b} \qquad \frac{\Delta \vdash a \quad \Delta \vdash b}{\Delta \vdash a \Rightarrow b}$$

$$\frac{\Delta \vdash a \quad \Delta \vdash j_i}{\Delta \vdash a \,!\, \{j_0, ..., j_n\}} \qquad \frac{\Delta, i : \varepsilon \vdash a}{\Delta \vdash \exists (i : \varepsilon).a}$$

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. 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.

T-SubVal **T-Var** **T-Unit** **T-Abs**

$$\frac{\Delta; \Gamma \vdash \nu : \tau_1 \quad \Delta \vdash \tau_2 \quad \tau_1 <: \tau_2}{\Delta; \Gamma \vdash \nu : \tau_2} \qquad \frac{(x : \tau) \in \Gamma}{\Delta; \Gamma \vdash x : \tau} \qquad \frac{}{\Delta; \Gamma \vdash () : ()} \qquad \frac{\Delta \vdash \tau_1 \quad \Delta; \Gamma, x : \tau_1 \vdash c : \underline{\tau_2}}{\Delta; \Gamma \vdash \lambda x.c : \tau_1 \to \underline{\tau_2}}$$

In the following rule
$h = handler(\nu)\ \{return\ x_r \to c_r, op_1(x_1; k_1) \to c_1, ..., op_n(x_n; k_n) \to c_n\}$.
T-Handler

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

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.

Explain...
T-SubComp

$$\frac{\begin{array}{l}\Delta; \Gamma \vdash c : \underline{\tau}_1 \\ \Delta \vdash \underline{\tau}_2 \\ \underline{\tau}_1 <: \underline{\tau}_2\end{array}}{\Delta; \Gamma \vdash c : \underline{\tau}_2}$$

T-Return

$$\frac{\Delta; \Gamma \vdash \nu : \tau}{\Delta; \Gamma \vdash return\ \nu : \tau \,!\, \varnothing}$$

T-App

$$\frac{\begin{array}{l}\Delta; \Gamma \vdash \nu_1 : \tau_1 \to \underline{\tau}_2 \\ \Delta; \Gamma \vdash \nu_2 : \tau_1\end{array}}{\Delta; \Gamma \vdash \nu_1\ \nu_2 : \underline{\tau}_2}$$

T-Handle

$$\frac{\begin{array}{l}\Delta, \overrightarrow{i}; \Gamma \vdash \nu : \underline{\tau}_1 \Rightarrow \underline{\tau}_2 \\ \Delta; \Gamma \vdash c : \exists \overrightarrow{i}.\underline{\tau}_1\end{array}}{\Delta; \Gamma \vdash with\ \nu\ handle\ c : \exists \overrightarrow{i}.\underline{\tau}_2}$$

Instances are checked to be in Δ.

T-Instance

$$\frac{(i : \varepsilon) \in \Delta}{\Delta; \Gamma \vdash \mathsf{inst}(i) : i}$$

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.

T-Op

$$\frac{\begin{array}{l}\Delta; \Gamma \vdash \nu_1 : i \\ (i : \varepsilon) \in \Delta \\ op \in O_\varepsilon \\ \Delta; \Gamma \vdash \nu_2 : \tau^1_{op} \\ \Delta; \Gamma, y : \tau^2_{op} \vdash c : \exists \overrightarrow{j}.\tau \,!\, r \\ i \in r\end{array}}{\Delta; \Gamma \vdash \nu_1 \# op(\nu_2; y.c) : \exists \overrightarrow{j}.\tau \,!\, r}$$

T-New

$$\frac{}{\Delta; \Gamma \vdash new\, \varepsilon : \exists (i : \varepsilon).i \,!\, \varnothing}$$

For sequencing we view both c_1 and c_2 as being annotated types with a possibly empty sequences of existential bindings in front, where the bindings are denoted $\overrightarrow{i}$ and $\overrightarrow{j}$. To typecheck c_2 all the bindings $\overrightarrow{i}$ are introduced in the context Δ. For the return type we concatenate all the bindings $\overrightarrow{i}$ and $\overrightarrow{j}$ and we take the union of the annotations r_1 of c_1 and the annotations r_2 of c_2.

T-Do

$$\frac{\begin{array}{l}\Delta; \Gamma \vdash c_1 : \exists \overrightarrow{i}.\tau_1 \,!\, r_1 \\ \Delta, \overrightarrow{i}; \Gamma, x : \tau_1 \vdash c_2 : \exists \overrightarrow{j}.\tau_2 \,!\, r_2\end{array}}{\Delta; \Gamma \vdash x \leftarrow c_1; c_2 : \exists \overrightarrow{i}.\exists \overrightarrow{j}.\tau_2 \,!\, (r_1 \cup r_2)}$$

1.6 Semantics

We give the small-step operational semantics for the system. These are the same as the one in [1] but with the instance creation computation added. In order to handle instance creation we update the relation to $c; n \rightsquigarrow c; n$ where c is the current computation and $n \in \mathbb{N}$ the id of the next instance that will be created.
For abstractions we have the usual beta-reduction rule.

$$\frac{}{(\lambda x.c)\ \nu\ ;\ i \rightsquigarrow c[\nu/x]\ ;\ i}$$

For instance creation we replace the call to *new* with the instance constant and we increase the instance id counter.

$$\frac{}{new\ E\ ;\ i \rightsquigarrow return\ \mathsf{inst}(i)\ ;\ i+1}$$

For sequencing we have three rules. The first rule reduces the first computation. The second rule substitutes the value of a return computation in the second computation. The last rule floats an operation call over the sequencing, this makes the handle computation easier, since we won't have to think about sequencing inside of a handle computation.

$$\frac{c_1\ ;\ i \rightsquigarrow c_1'\ ;\ i'}{(x \leftarrow c_1; c_2)\ ;\ i \rightsquigarrow (x \leftarrow c_1'; c_2)\ ;\ i'}$$

$$\frac{}{(x \leftarrow return\ \nu; c)\ ;\ i \rightsquigarrow c[\nu/x]\ ;\ i}$$

$$\frac{}{(x \leftarrow \mathsf{inst}(j)\#op(\nu; y.c_1); c_2)\ ;\ i \rightsquigarrow \mathsf{inst}(j)\#op(\nu; y.(x \leftarrow c_1; c_2))\ ;\ i}$$

In the following rules
$h = handler(\mathsf{inst}(j))\ \{return\ x_r \rightarrow c_r, op_1(x_1; k_1) \rightarrow c_1, ..., op_n(x_n; k_n) \rightarrow c_n\}$.
We can reduce the computation that we want to handle.

$$\frac{c\ ;\ i \rightsquigarrow c'\ ;\ i'}{with\ h\ handle\ c\ ;\ i \rightsquigarrow with\ h\ handle\ c'\ ;\ i'}$$

If we are handling a return computation we simply substitute the value in the return case of the handler.

$$\frac{}{with\ h\ handle\ (return\ \nu)\ ;\ i \rightsquigarrow c_r[\nu/x_r]\ ;\ i}$$

If we are handling an operation call where the instance matches the instance of the handler and the operation is in the handler then we can reduce to the corresponding operation clause with the parameter value and the continuation substituted. Note that we nest the handle computation inside of the continuation, this describes deep handlers.

$$\frac{op_i \in \{op_1, ..., op_n\}}{\mathit{with}\ h\ \mathit{handle}\ (\mathsf{inst}(j)\#op_i(\nu; x.c))\ ;\ i \rightsquigarrow c_i[\nu/x_i, (\lambda x.\mathit{with}\ h\ \mathit{handle}\ c)/k_i]\ ;\ i}$$

If the instance is different from the instance in the handler or the operation is not in the handler then we float the operation call over the handling computation.

$$\frac{op \notin \{op_1, ..., op_n\} \vee k \neq j}{\mathit{with}\ h\ \mathit{handle}\ (\mathsf{inst}(k)\#op(\nu; x.c))\ ;\ i \rightsquigarrow \mathsf{inst}(k)\#op(\nu; x.\mathit{with}\ h\ \mathit{handle}\ c)\ ;\ i}$$

1.7 Theorems

The most important theorem is the type soundness theorem, which states that any computation that typechecks will not get stuck. This is proven using two auxiliary theorems preservation and progress.
The preservation theorem states that for any program that typechecks, if we take a step using the semantics then the resulting program will still typecheck with the same type.
Theorem 3 (Preservation)**.** *If* $\Delta; \Gamma \vdash c : \underline{\tau}$ *and* $c \rightsquigarrow c'$ *then* $\Delta; \Gamma \vdash c' : \underline{\tau}$

The progress theorem states that any computation that typechecks is either able to take a step, is a value.
In order to proof this we have the auxiliary theorem for effectful computation where the evaluation can get stuck on operation calls.
Theorem 4 (Progress effectful)**.** *If* $\cdot; \cdot \vdash c : \underline{\tau}$ *then either:*

- $c \rightsquigarrow c'$ *for some* c'
- $c = return\ \nu$ *for some* ν
- $c = \mathsf{inst}(i)\#op(\nu; x.c)$ *for some* i, op, ν *and* c

Note that the progress theorem requires that the computation has no effects ($!\varnothing$), else the computation could get stuck on an operation call.
Theorem 5 (Progress)**.** *If* $\cdot; \cdot \vdash c : \exists \overrightarrow{i}.\tau\ !\ \varnothing$ *then either:*

- $c \rightsquigarrow c'$ *for some* c'
- $c = return\ \nu$ *for some* ν

Using preservation and progress we can proof type soundness (also called type safety). Here also we require that the computation has no effects.
Theorem 6 (Type soundness)**.** *If* $\cdot; \cdot \vdash c : \exists \overrightarrow{i}.\tau\ !\ \varnothing$ *and* $c, i \rightsquigarrow^* c', i'$ *(where* $c', i' \not\rightsquigarrow c'', i''$*) then* $c' = return\ \nu$ *for some* ν

The determinism theorem states that any computation has a single evaluation path.
Theorem 7 (Determinism)**.** *If* $c \rightsquigarrow c'$ *and* $c \rightsquigarrow c''$ *then* $c' = c''$

1.8 Examples

We will show some examples together with the types that the discussed type system will assign to them. After each example is a fitch-style type derivation. 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 ()
```

1	$\mid \_ : ()$	
2	$new\ Flip : \exists(j : Flip).i\ !\ \varnothing$	
3	$i : E \mid inst : i$	
4	$\mid x : Bool$	
5	$return\ x : Bool\ !\ \varnothing$	T-Return, 4
6	$return\ x : Bool\ !\ \{i\}$	T-SubComp, 3, 5
7	$inst\#flip((); x.return\ x) : Bool\ !\ \{i\}$	T-Op, 6
8	$inst \leftarrow new\ Flip; inst\#flip((); x.return\ x) : \exists(i : Flip).Bool\ !\ \{i\}$	T-Do, 2, 7
9	$\lambda\_.inst \leftarrow new\ Flip; inst\#flip((); x.return\ x) : () \rightarrow \exists(i : Flip).Bool\ !\ \{i\}$	T-Abs, 8

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 handler(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;
  return ()
```

1	$\mid \_ : ()$	
2	$new\ Flip : \exists(j : Flip).i \mathbin{!} \varnothing$	
3	$i : E \mid inst : i$	
4	$() : ()$	
5	$return\ () : () \mathbin{!} \varnothing$	T-Return, 4
6	$inst \leftarrow new\ Flip; return\ () : \exists(i : Flip).() \mathbin{!} \varnothing$	T-Do, 2, 5
7	$inst \leftarrow new\ Flip; return\ () : () \mathbin{!} \varnothing$	T-ExistsRemove, 6
8	$\lambda\_.inst \leftarrow new\ Flip; return\ () : () \rightarrow () \mathbin{!} \varnothing$	T-Abs, 7

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

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

In the following function *nested* we create a new instance and handle it, but in the computation we want to handle another instance is created and used. Only one operation call will be handled by the handler, the one called on $i1$. The operation call made on $i2$ is unhandled and so we are left with one existential quantifier.

```
nested : () -> exists (i:Flip). Bool!{i}
nested _ =
  i1 <- new Flip;
  with handler(i1) {
    flip _ k -> k True
  } handle (
    i2 <- new Flip;
    x <- i1#flip ();
    y <- i2#flip ();
    return (x && y)
  )
```

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.