May 4, 2016

1 Effects

Fix some set of resources R. A resource is some language primitive that has the authority to directly perform I/O operations. Elements of the set R are denoted by r. Π is a fixed set of operations on resources. Its members are denoted π. An effect is a member of the set of pairs $R \times \Pi$. A set of effects is denoted by ε. In this system we cannot dynamically create resources or resource-operations.

Throughout we refer to the notions of effects and captures. A piece of code C has the effect (r, π) if operation π is performed on resource r during execution of C. C captures the effect (r, π) if it has the authority to perform operation π on resource r at some point during its execution.

We use $r.\pi$ as syntactic sugar for the effect (r, π). For example, $FileIO.append$ instead of $(FileIO, append)$.

Types are either resources or structural. Structural types have a set of method declarations. An object of a particular structural type $\{\bar{\sigma}\}$ can have any of the methods defined by σ invoked on it. The structural type $\varnothing$ with no methods is called `Unit`.

We assume there are constructions of the familiar types using the basic structural type $\varnothing$ and method declarations (for example, $\mathbb{N}$ could be made using $\varnothing$ and a `successor` function, Peano-style).

Note the distinction between methods (usually denoted m) and operations (usually denoted π). An operation can only be invoked on a resource; resources can only have operations invoked on them. A method can only be invoked on an object; objects can only have methods invoked on them.

We make a simplifying assumption that every method/lambda takes exactly one argument. Invoking some operation π on a resource returns $\varnothing$.

2 Fully-Annotated Programs

In this first system every method in the program is explicitly annotated with its set of effects.

2.1 Grammar

$$
\begin{array}{rcll}
e & ::= & x & \textit{expressions} \\
 & | & r & \\
 & | & \texttt{new}\ x \Rightarrow \overline{\sigma = e} & \\
 & | & e.m(e) & \\
 & | & e.\pi(e) & \\
 & & & \\
\tau & ::= & \{\bar{\sigma}\} \mid \{\bar{r}\} & \textit{types} \\
 & & & \\
\sigma & ::= & d\ \texttt{with}\ \varepsilon & \textit{labeled decls.} \\
 & & & \\
d & ::= & \texttt{def}\ m(x : \tau) : \tau & \textit{unlabeled decls.} \\
 & & & \\
\Gamma & ::= & \varnothing & \\
 & | & \Gamma,\ x : \tau &
\end{array}
$$

Notes:

- Declarations (σ-terms) are annotated by what effects they have.
- d-terms do not appear in programs, except as part of σ-terms.
- All methods (and lambda expressions) take exactly one argument. If a method specifies no argument, then the argument is implicitly of type. `Unit`.
- Although $e_1.\pi(e_2)$ is a syntactically valid expression, it is only well-formed if e_1 is a resource (so e_1 is only a resource in well-typed programs).

2.2 Syntactic Sugar

Programs may also contain `let`, `val`, and λ expressions. These can be encoded using the current grammar using transformation rules. We use $\rightsquigarrow$ for this purpose: if the relation $\Gamma \mid a \rightsquigarrow b$ holds if and only if in any piece of code $S[a]$ with context Γ, the dynamic semantics of $S[b/a]$ are exactly the same as $S[a]$.

$$
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \rightsquigarrow (\texttt{new def}\ m(x : \tau_1) : \tau_2)).m(e_1)}\ (\textsc{Trans-Let})
$$

$$
\frac{\Gamma \vdash e_1 : \tau_1}{\texttt{val}\ x : \tau_1 = e_1 \rightsquigarrow \texttt{let}\ \alpha x = (\texttt{new def}\ get() : \tau_1 = e_1)\ \texttt{in}\ (\alpha x.get())}\ (\textsc{Trans-ValDef})
$$

$$
\frac{x : \tau \in \Gamma}{x \rightsquigarrow \alpha x.get()}\ (\textsc{Trans-Val})
$$

Notes:

- We use the symbol α to prefix anonymous objects. These are objects constructed by the application of transformation rule. The variable x is turned into αx, an object with a single get method that returns the expression defining x.

2.3 Rules

$\boxed{\Gamma \vdash e : \tau \ \texttt{with} \ \varepsilon}$

$$\frac{}{\Gamma,\ x : \tau \vdash x : \tau \ \texttt{with} \ \varnothing} \ (\varepsilon\text{-}\textsc{Var}) \qquad \frac{}{\Gamma,\ r : \{r\} \vdash r : \{r\} \ \texttt{with} \ \varnothing} \ (\varepsilon\text{-}\textsc{Resource})$$

$$\frac{\Gamma,\ x : \{\bar{\sigma}\} \vdash \overline{\sigma = e} \ \texttt{OK}}{\Gamma \vdash \texttt{new} \ x \Rightarrow \overline{\sigma = e} : \{\bar{\sigma}\} \ \texttt{with} \ \varnothing} \ (\varepsilon\text{-}\textsc{NewObj})$$

$$\frac{\Gamma \vdash e_1 : \{\bar{r}\} \ \texttt{with} \ \varepsilon_1 \quad \Gamma \vdash e_2 : \tau_2 \ \texttt{with} \ \varepsilon_2 \quad \pi \in \Pi}{\Gamma \vdash e_1.\pi(e_2) : \varnothing \ \texttt{with} \ \{\bar{r}, m\} \cup \varepsilon_1 \cup \varepsilon_2} \ (\varepsilon\text{-}\textsc{OperCall})$$

$$\frac{\Gamma \vdash e_1 : \{\bar{\sigma}\} \ \texttt{with} \ \varepsilon_1 \quad \Gamma \vdash e_2 : \tau_2 \ \texttt{with} \ \varepsilon_2 \quad \sigma_i = \texttt{def} \ m_i(y : \tau_2) : \tau \ \texttt{with} \ \varepsilon}{\Gamma \vdash e_1.m_i(e_2) : \tau \ \texttt{with} \ \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon} \ (\varepsilon\text{-}\textsc{MethCallObj})$$

$\boxed{\Gamma \vdash \sigma = e \ \texttt{OK}}$

$$\frac{\Gamma,\ x : \tau \vdash e : \tau' \ \texttt{with} \ \varepsilon \quad \sigma = \texttt{def} \ m(x : \tau) : \tau' \ \texttt{with} \ \varepsilon}{\Gamma \vdash \sigma = e \ \texttt{OK}} \ (\varepsilon\text{-}\textsc{ValidImpl}_\sigma)$$

Notes:

- Every expression in the program must be explicitly annotated; either as σ-terms or by what they capture.
- The rules ε-Var, ε-Resource, and ε-NewObj have in their consequents an expression typed with no effect: merely having an object or resource is not an effect; you must do something with it, like a call a method on it, in order for it to be an effect.
- ε-ValidImpl says that the return type and effects of the body of a method must agree with what its signature says.
- In ε-MethCallResource, we may only call a method m on a resource r if m is a predefined operation in the set M. Invoking m returns the resource r you called it upon (which has potentially different state afterwards).

3 Partially-Annotated Programs

In this second system methods may either be fully labeled with their effects or have no labels. When they have no labels a conservative effect inference is performed using rules which provide an upper-bound (not necessarily tight) on the effects of the code when executed.

3.1 Grammar

$$
\begin{array}{rcll}
e & ::= & x & \textit{expressions} \\
& | & r & \\
& | & \texttt{new}_\sigma\ x \Rightarrow \overline{\sigma = e} & \\
& | & \texttt{new}_d\ x \Rightarrow \overline{d = e} & \\
& | & e.m(e) & \\
& | & e.\pi(e) & \\
\\
\tau & ::= & \{\bar{\sigma}\} & \textit{types} \\
& | & \{\bar{r}\} & \\
& | & \{\bar{d}\} & \\
& | & \{\bar{d}\ \texttt{captures}\ \varepsilon\} & \\
\\
\sigma & ::= & d\ \texttt{with}\ \varepsilon & \textit{labeled decls.} \\
\\
d & ::= & \texttt{def}\ m(x : \tau) : \tau & \textit{unlabeled decls.}
\end{array}
$$

Notes:

- σ denotes a declaration with effect labels. d denotes a declaration without effect labels.
- There are two new expressions: $\texttt{new}_\sigma$ for objects whose declarations are annotated; $\texttt{new}_d$ for objects whose declarations aren't.
- $\{\bar{d}\ \texttt{captures}\ \varepsilon\}$ is a special kind of type that doesn't appear in the source program, but may be assigned as a consequence of the capture rules.

3.2 Rules

In addition to the rules from the previous system, the partially-annotated system has the following rules.

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma,\ x : \tau \vdash x : \tau}\ (\text{T-Var}) \qquad \frac{}{\Gamma,\ r : \{\bar{r}\} \vdash r : \{\bar{r}\}}\ (\text{T-Resource})$$

$$\frac{\Gamma \vdash r : \{\bar{r}\} \quad \Gamma \vdash e : \tau \quad m \in M}{\Gamma \vdash r.\phi(e_1) : \varnothing}\ (\text{T-MethCall}_r)$$

$$\frac{\Gamma \vdash e_1 : \{\bar{\sigma}\},\ \texttt{def}\ m(x : \tau_1) : \tau_2\ \texttt{with}\ \varepsilon \in \{\bar{\sigma}\} \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1.m(e_2) : \tau_2}\ (\text{T-MethCall}_\sigma)$$

$$\frac{\Gamma \vdash e_1 : \{\bar{d}\},\ \texttt{def}\ m(x : \tau_1) : \tau_2 \in \{\bar{d}\} \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1.m(e_2) : \tau_2}\ (\text{T-MethCall}_d)$$

$$\frac{\Gamma \vdash \sigma_i = e_i\ \texttt{OK}}{\Gamma \vdash\ \texttt{new}_\sigma\ x \Rightarrow \overline{\sigma = e} : \{\bar{\sigma}\}}\ (\text{T-New}_\sigma) \qquad \frac{\Gamma \vdash d_i = e_i\ \texttt{OK}}{\Gamma \vdash\ \texttt{new}_d\ x \Rightarrow \overline{d = e} : \{\bar{d}\}}\ (\text{T-New}_d)$$

$\boxed{\Gamma \vdash d = e \ \texttt{OK}}$

$$\frac{d = \texttt{def}\ m(x : \tau_1) : \tau_2 \quad \Gamma \vdash e : \tau_2}{\Gamma \vdash d = e\ \texttt{OK}} \ (\varepsilon\text{-}\textsc{ValidImpl}_d)$$

$\boxed{\Gamma \vdash e : \tau\ \texttt{with}\ \varepsilon}$

$$\frac{\varepsilon = effects(\Gamma') \quad \Gamma' \subseteq \Gamma \quad \Gamma', x : \{\bar{d}\ \texttt{captures}\ \varepsilon\} \vdash \overline{d = e}\ \texttt{OK}}{\Gamma \vdash \ \texttt{new}_d\ x \Rightarrow \overline{d = e} : \{\bar{d}\ \texttt{captures}\ \varepsilon\}\ \texttt{with}\ \varnothing} \ (\text{C-}\textsc{NewObj})$$

$$\frac{\Gamma \vdash e_1 : \{\bar{d}\ \texttt{captures}\ \varepsilon\}\ \texttt{with}\ \varepsilon_1 \quad \Gamma \vdash e_2 : \tau_2\ \texttt{with}\ \varepsilon_2 \quad d_i := \ \texttt{def}\ m_i(y : \tau_2) : \tau}{\Gamma \vdash e_1.m_i(e_2) : \tau\ \texttt{with}\ \varepsilon_1 \cup \varepsilon_2 \cup effects(\tau_2) \cup \varepsilon} \ (\text{C-}\textsc{MethCall})$$

Notes:

- The ε judgements are to be applied to annotated parts of the program; the C rules for unannotated parts.
- The rules ε-Var, ε-Resource, and ε-NewObj have in their antecedents an expression typed with no effect. Merely having an object or resource is not an effect; you must do something with it, like a call a method on it, in order for your program to have effects.
- The T judgements before standard typechecking, but they operate on annotated terms. They are needed to apply the ε-ValidImpl$_d$) rule.
- In applying C-NewObj the variable Γ is the current context. The variable Γ' is some sub-context. A good choice of sub-context is Γ restricted to the free variables in the method-body being typechecked. This means we only consider the effects used in the method-body and gives a better approximation of its effects.
- When an unannotated d-declaration is encountered it is first assigned a γ-type by C-NewObj. This annotates it as capturing a certain set of effects. C-MethCall can then conclude its effects to be what it captures.

3.3 Effects Function

The `effects` function returns the set of effects in a particular typing context.

A method m can return a resource r (directly or via some enclosing object). Returning a resource isn't an effect but it means any unannotated program using m also captures r. To account for this, when the `effects` function is operating on a type τ it must analyse the return type of the method declarations in τ.

- $\texttt{effects}(\varnothing) = \varnothing$
- $\texttt{effects}(\{\bar{r}\}) = \{(r, m) \mid r \in \bar{r}, m \in M\}$
- $\texttt{effects}(\{\bar{\sigma}\}) = \bigcup_{\sigma \in \bar{\sigma}} \ \texttt{effects}(\sigma)$
- $\texttt{effects}(\{\bar{d}\}) = \bigcup_{d \in \bar{d}} \ \texttt{effects}(d)$
- $\texttt{effects}(d\ \texttt{with}\ \varepsilon) = \varepsilon \cup \texttt{effects}(d)$
- $\texttt{effects}(\texttt{def m}(x : \tau_1)\ \tau_2) = \texttt{effects}(\tau_2)$