May 4, 2016

1 Example 1

This example is a fully-annotated program. We can check it using rules from the fully-annotated system.

```
1  // Γ0 = {FileIO : {FileIO}}
2  let logger1 = new
3      def log(entry : String) : Unit with FileIO.append
4        FileIO.append('/logs/mylog.txt', entry)
5
6  // Γ1 = {FileIO : {FileIO}, logger1 : {log : String → Unit}}
7  in new
8     def main() : Unit with FileIO.append
9       logger1.log('Hello, World!')
```

Start with Γ_0. After execution of line 2, we obtain Γ_1. Line 7 declares an unannotated object type so we want to match it with the consequent in ε-NewObj.

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

To prove $\overline{\sigma = e}$ OK we need the following rules.

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

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

`logger1.log(″Hello, world!″)` can be checked with ε-MethCallResource. In this case, $logger1 : \{...\}$ `with` $\varnothing$ by ε-Var and `″Hello, world!″` : *String* `with` $\varnothing$ (although there's no rule for constants).

The definition of `log` says that it has the effect `FileIO.append`, so the effect set for `logger1.log(″Hello, world!″)` is the singleton {`FileIO.append`}.

With the body of `main` typechecked we can apply ε-ValidImpl$_\sigma$, because the annotation for `main` matches the effect we computed for its body. Then we know that the method implementations for the new object are well-formed.

Finally we may apply ε-NewObj. We conclude that the type is {`main : Unit → Unit with {FileIO.append}`} .

2 Example 2

This example is like the previous one but the main object is not annotated. So we need to use the capture-rules from the partially-annotated system.

```
1  // Γ0 = {FileIO : {FileIO}}
2  let logger1 = new
3      def log(entry : string) : Unit with FileIO.append
4        FileIO.append('/logs/mylog.txt', entry)
5
6  // Γ1 = {FileIO : {FileIO}, logger1 : {log : String → Unit}}
7  in new
8     def main() : Unit
9       logger1.log('Hello, World!')
```

1. Start with Γ_0. After execution of line 2, we obtain Γ_1. Line 7 declares an unannotated object type so we want to match that with the consequent of C-NewObj.

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

Typechecking

2. We must type the body of the `main` method. First we type `logger1`. As $\texttt{logger1} \in \Gamma$ we can apply T-Var.

3. There is no rule for typechecking string constants but $''\texttt{Hello}, \texttt{world}''$ should typecheck to `String`.

4. With 1. and 2. We can typecheck $\texttt{logger1.log}(''\texttt{Hello}, \texttt{world}'')$ with T-MethCall$_\sigma$. All the types match up, so this expression types to `Unit`. The declared return type of `main` is also `Unit`, so we're good.

Effect-Checking

5. We need the `effects` function and a choice of Γ'. We choose $\Gamma' = \{\texttt{logger1} : \{\texttt{log} : \texttt{Str} -> \texttt{Unit}\}\}$, because `logger1` is the only free variable appearing in the body of `main`. We need the following cases of the effects function.

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

6. By applying those cases of the $effects$ function we see that:
$\texttt{effects}(\Gamma')$
$= \texttt{effects}(\texttt{logger1})$
$= \texttt{effects}(\texttt{logger1.log})$
$= \texttt{effects}(\texttt{def log}(\texttt{entry} : \texttt{string}) : \texttt{Unit with FileIO.append}$
$= \{\texttt{FileIO.append}\} \cup \texttt{effects}(\texttt{Unit})\}$
$= \{\texttt{FileIO.append}\} \cup \texttt{effects}(\varnothing)\}$
$= \{\texttt{FileIO.append}\} \cup \varnothing\}$
$= \{\texttt{FileIO.append}\}$

Conclusion

7. Now we've satisfied the antecedents of C-NewObj. We label the new object with the following type:
$\texttt{main} : \texttt{Unit} \rightarrow \texttt{Unit captures}\ \{\texttt{FileIO.append}\}$.

3 Example 3

In this example the logger exposes the `FileIO` resource through a method, so anyone who calls that resource will capture every effect on `FileIO`.

```
1  // Γ0 = {FileIO : {FileIO}}
2  let logger2 = new
3     def log(entry : String) : Unit with FileIO.append
4        FileIO.append('/logs/mylog.txt', entry)
5     def expose() : { FileIO } with ∅
6        FileIO
7
8   // Γ1 = {FileIO : {FileIO}, logger2 : {log : String → Unit,  expose : Unit → FileIO}}
9  in new
10    def main() : Unit
11       logger2.expose().read('/etc/passwd')   // has a read effect that is not captured
```

1. Similar to example 2 we want to apply C-NewObj.

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

Type-Checking

2. To type the body of `main` we apply T-MethCall$_\sigma$ to `logger2.expose()`, which types to {`FileIO`}. Then we can type `logger2.expose().read("/etc/passwd")` by applying T-MethCall$_r$, which says that it types to $\varnothing$ = `Unit`. This matches the return type of `main`, so we're good.

Effect-Checking

3. Our choice of Γ' will be `logger2`, as this is the set of free variables in the body of `main`. We use the following cases of the `effects` function.

- `effects`$(\varnothing) = \varnothing$
- `effects`$(\{\bar{r}\}) = \{(r, m) \mid r \in \bar{r}, m \in M\}$
- `effects`$(d$ `with` $\varepsilon) = \varepsilon \cup$ `effects`(d)
- `effects`(`def` `m`$(x : \tau_1)\ \tau_2)$ = `effects`(τ_2)
- `effects`$(\{\bar{\sigma}\}) = \bigcup_{\sigma \in \bar{\sigma}}\ effects(\sigma)$

4. `effects(logger2.log)`
= `effects(def log(entry : String) : Unit with FileIO.append)`
= {`FileIO.append`} ∪ `effects(Unit)`
= {`FileIO.append`}

5. `effects(logger2.expose)`
= `effects(def expose() : {FileIO} with` $\varnothing$`)`
= $\varnothing$ ∪ `effects({FileIO})`
= {(`FileIO`, `m`) | `m` ∈ `M`}
= {`FileIO.append`, `FileIO.write`, `FileIO.read`}

6. By combining 4. and 5. and the last case above of the `effects` function: `effects(logger2)`
= `effects(logger2.log)` ∪ `effects(logger2.expose)`
= {`FileIO.append`} ∪ {`FileIO.append`, `FileIO.write`, `FileIO.read`}
= {`FileIO.append`, `FileIO.write`, `FileIO.read`}

Conclusion

7. The result of 2. and 6. show the antecedents of C-NewObj hold. We can apply the consequence, typing the newly-created object to the following type.

{`main : Unit → Unit captures {FileIO.append, FileIO.read, FileIO.write} with` $\varnothing$}

4 Example 4

In this example the `FileIO` resource is exposed by returning an object with an authority for it.

```
1  type SigFoo
2     def getIO() : { FileIO } with ∅
3
4  let logger3 = new
5     def log(entry : String) : Unit with FileIO.append
6        FileIO.append('/logs/mylog.txt', entry)
```

```
7     def expose() : SigFoo with ∅
8        new
9           def getIO() : { FileIO } with ∅
10             FileIO
11
12  in new
13     def main() : Unit
14        logger3.expose().getIO().read('/etc/passwd')
```

1. As in previous examples we want to apply C-NewObj.

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

Typechecking

First we'll typecheck the body of `main`.

2. `logger3` : {`log` : `String` → `Unit`..., `expose` : `Unit` → `SigFoo`...} by the rule T-Var, as `logger3` ∈ Γ.

3. Apply T-MethCall$_\sigma$ to `logger3.expose()`. The argument is of type `Unit` (need a rule for this? – no argument type specified). The return type of `expose` is `SigFoo`, so `logger3.expose()` : `SigFoo`.

4. Apply T-MethCall$_\sigma$ to `logger3.expose().getIO()`. This typechecks to $\{FileIO\}$.

5. Apply T-MethCall$_r$ to `logger3.expose().getIO().read(''/etc/passwd'')`. This typechecks to ∅ which matches the declared return type of `main`. Then the definition of `main` is well-typed.

Effect-Checking

6. The free variables of `main` is `logger3`, so we choose Γ' containing only `logger3`. Here are the relevant cases for the `effects` function.

- `effects`(∅) = ∅
- `effects`($\{\bar{r}\}$) = $\{(r, m) \mid r \in \bar{r}, m \in M\}$
- `effects`($\{\bar{\sigma}\}$) = $\bigcup_{\sigma \in \bar{\sigma}}$ `effects`(σ)
- `effects`($\{\bar{d}\}$) = $\bigcup_{d \in \bar{d}}$ `effects`(d)
- `effects`($\{\bar{d}$ `captures` $\varepsilon_1\}$ `with` ε_2) = $\varepsilon_1 \cup \varepsilon_2$
- `effects`(d `with` ε) = $\varepsilon \cup$ `effects`(d)
- `effects`(`def m`($x : \tau_1$) τ_2) = `effects`(τ_2)

7. `effects`(`logger3.log`)
= `effects`(`def log(entry : String) : Unit with FileIO.append`)
= {`FileIO.append`} ∪ `effects`(`Unit`)
= {`FileIO.append`}

8. `effects`(`SigFoo`)
= `effects`(`getIO`)
= `effects`(`def getIO() : {FileIO} with` ∅
= ∅ ∪ `effects`({`FileIO`})
= {`FileIO.append`, `FileIO.read`, `FileIO.write`}

9. `effects`(`logger3.expose`)
= `effects`(`def expose() : SigFoo with` ∅
= ∅ ∪ `effects`(`SigFoo`)
= {`FileIO.append`, `FileIO.read`, `FileIO.write`} (by 8)

10. `effects(logger3)`
= `effects(logger3.log)` ∪ `effects(logger3.expose)`
= {`FileIO.append`, `FileIO.read`, `FileIO.write`} (by 7 and 9)

Conclusion

11. The results of 10 and 5 satisfy the antecendets of C-NewObj. Applying it we conclude that the object types to: {`main` : `Unit` → `Unit captures` {`FileIO.read`, `FileIO.write`, `FileIO.append`}} `with` ∅

5 Example 5

This is an example with parametricity.

```
1  // Γ0 = {{FileIO}}
2  type SigPasswordReader
3     def readPasswords(fileio : { FileIO }) : String with FileIO.read
4  let passwordReader = new
5     def readPasswords(fileio : { FileIO }) : String with FileIO.read
6        fileio. read('/etc/passwd')
7  in
8     let logger4 = new
9        def log(entry : String) : Unit with FileIO.append
10          FileIO.append('/log/mylog.txt', entry)
11       def enablePasswordReading(pr : SigPasswordReader) : Unit
12          pr.readPasswords(FileIO)
13    in new
14       def main() : Unit
15          logger4.enablePasswordReading(passwordReader)
16 /* This example also illustrates parametricity: passwordReader accepts any resources of type { FileIO } */
```

1. Want to apply C-NewObj.

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

Type-Checking

2. `passwordReader` ∈ Γ so we can apply T-Var. `passwordReader` types to `SigPasswordReader`.

3. Apply T-MethCall$_\sigma$ to `logger4.enablePasswordReading(passwordReader)`. This types to `Unit`, which matches the declared return type of `main`, so everything is well-typed.

Effect-Checking

4. `logger4` and `passwordReader` occur free in the body of the new object, so we choose Γ' equal to Γ, restricted to those two variables. The relevant cases of the `effects` function are:

- $\texttt{effects}(\varnothing) = \varnothing$
- $\texttt{effects}(\{\bar{r}\}) = \{(r, m) \mid r \in \bar{r}, m \in M\}$
- $\texttt{effects}(\{\bar{\sigma}\}) = \bigcup_{\sigma \in \bar{\sigma}} \texttt{effects}(\sigma)$
- $\texttt{effects}(\{\bar{d}\}) = \bigcup_{d \in \bar{d}} \texttt{effects}(d)$
- $\texttt{effects}(\{\bar{d} \texttt{ captures } \varepsilon_1\} \texttt{ with } \varepsilon_2) = \varepsilon_1 \cup \varepsilon_2$
- $\texttt{effects}(d \texttt{ with } \varepsilon) = \varepsilon \cup \texttt{effects}(d)$
- $\texttt{effects}(\texttt{def m}(x : \tau_1)\ \tau_2) = \texttt{effects}(\tau_2)$

5. $\texttt{effects}(\Gamma') = \texttt{effects}(\texttt{logger4}) \cup \texttt{effects}(\texttt{passwordReader})$, so we'll compute the effects of `logger4` and `passwordReader` first.

6. effects(logger4)
= effects(logger4.log) ∪ effects(logger4.enablePasswordReading).

6.1. First we have
effects(logger4.log)
= effects(def log(entry : String) : Unit with FileIO.append)
= {FileIO.append} ∪ effects(Unit)
= {FileIO.append}

6.2. Second we have
effects(logger4.enablePasswordReading)
= effects(def enablePasswordReading(pr : SigPasswordReader) : Unit with ∅)
= ∅ ∪ effects(Unit)
= ∅

6.3. effects(logger4) = {FileIO.append} (6.1 and 6.2).

7. effects(passwordReader)
= effects(passwordReader.readPasswords)
= effects(def readPasswords(fileio : {FileIO}) : String with FileIO.read)
= {FileIO.read} ∪ effects(String)
= {FileIO.read}

8. effects(Γ′)
= effects(logger4) ∪ effects(passwordReader)
= {FileIO.append} ∪ {FileIO.read} (from 6. and 7.)
= {FileIO.append, FileIO.read}

Conclusion

9. From 3. and 8. we may apply C-NewObj. The object created has the following type.

{main : Unit → Unit captures {FileIO.read, FileIO.append}} with ∅

6 Example 6

This example looks at a function which takes a function f_1 as argument and returns another function f_2. f_1 has effects, causing f_2 to have effects.

```
1
2  let env = new
3      def filter (f : (Int → Bool) → Int) : Int → Bool with ∅
4          λx : Int . f(x)
5      def isZero (x : Int) : Bool with FileIO.append
6          let _ = FileIO.append(x) in
7              x == 0
8  in new
9      def main (x : Int) : Bool
10         env.filter(env.is_zero,x)
```

1. Want to apply C-NewObj.

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

2. Straightforward application of the rule. The desired effect (FileIO) is captured because it is captured by env, which is in Γ'.

7 Example 7

This example looks at a function which takes a function f_1 as argument and returns another function f_2. f_1 is pure, but f_2 has effects.

```
1
2  let env = new
3      def filter (f : Int → Bool) : Int → Bool with FileIO.append
4          λx : Int . let _ = FileIO.append(x)
5                   in f(x)
6      def isZero (x : Int) : Bool with ∅
7          x == 0
8  in new
9      def main (x : Int) : Bool
10         let f = env.filter(env.is_zero)
11             in f(x)
```

We're still safe because `FileIO` is captured by `env`, which is in Γ', so we'll capture it in $effects(\Gamma')$.

8 Example 8

```
1  let obj1 = new
2      def fmake () : Int → Bool with FileIO.append
3          λx : Int . let _ = FileIO.append(x)
4              in x == 0
5
6  in let obj2 = new
7      def app (f : Int → Bool,x : Int) : Bool with ∅
8          f(x)
9
10 in new
11     def main () : Unit
12         let f = obj1.fmake()
13             in obj2.app(f, 3)
```

This one is OK. Although the effect actually takes place during execution of $obj2$, it is captured by the client calling $obj2$ because $obj1$ creates the effectful function and that's in the environment when the client is executing.

Invoking a higher-order function hof may have an effect not captured by that higher-order function, because the client passed in some function f with effects.

If the client constructed f, and f has effects, then they must be effects on the resources in the context visible to the client, and so any client code involving $hof(f)$ will have the effects in f captured.

If the client obtains a function f with effects from some object o, via a method $o.m$ which builds and returns f, then $o.m$ (and by extension, o) will capture all effects in f. Since o is visible to the client, then o is in the environment, so the effects of f are captured.

9 Example 9

This has partially-labeled declarations.

```
1  // Γ0 = {{FileIO}}
2  let logger2 = new
3     def log(entry : String) : Unit with FileIO.append
4       FileIO.append('/logs/mylog.txt', entry)
5     def expose() : { FileIO }
6       FileIO
7
```

```
8    // Γ1 = {{FileIO}, logger2}
9  in new
10   def main() : Unit
11     logger2.expose().read('/etc/passwd')  // has a read effect that is not captured
```