Practical Foundations for Programming Languages

Jason Hu, Zhe Zhou, Ramana Nagasamudram

1 Dynamic Classification

Each summand is a class of values (sometimes called run-time types, but this makes it confusing). *Dynamic dispatch* is just a case analysis for a sum type. There is a discussion in PFPL about dynamic dispatch (to dispel certain assumptions ; discussion of a *dispatch matrix*). When you do case analysis of some element of a sum type, you recover information because you know what the underlying data is. This is an important idea – case analysis reveals information.

What we want: Some type of "extensible" sum. That is a sum of a bunch of things, e.g. $\tau_1 + \tau_2 + \dots$ (we want the $\dots$ to be part of the syntax). We want dynamical extensibility. At run-time, we are generating new classes of data that we can dispatch on. We have things that are dynamically new (as opposed to statically new). This means that it will be completely tied with issues of information flow or knowledge flow.
A dynamically new class corresponds to an "unguessable" secret (or perfect encryption by α-conversion). "Encryption is just α-conversion and Church invented it in the "30s". It induces *capabilities* express *integrity* and *confidentiality*.
Key idea: exceptions (i.e. the values associated with an exception) are "shared secrets". Raiser ensures the integrity of the data, and you allow the handler to violate the confidentiality of the data.

2 FPC (by-value) example

We assume that there is a exception control mechanism.

$$\begin{aligned} \tau &::= \cdots \mid \texttt{clsfd} \mid \tau\ \texttt{cls} \\ v &::= \cdots \mid \texttt{in}[c](v) \mid \&c \\ m &::= \cdots \mid \texttt{isin}[c](v; x.m_1; m_2) \mid \texttt{new}[\tau](c.m) \mid \texttt{reveal}(v; c.m) \end{aligned}$$

`clsfd` is dynamically classified – we are going to attach a class to a data item, and we can generate these classes at runtime. The classes c are an indexing scheme (that happens to be open-ended). τ `cls` is a "class reference". (The book contains a section on references that has nothing to do with state). $\texttt{cls}[c]$ is written as $\&c$.

$$\Gamma \vdash_\Sigma v : \tau \qquad\qquad \Gamma \vdash_\Sigma m \mathrel{\dot{\sim}} \tau$$

$\Sigma = c_1 \sim \tau_1, \dots, c_n \sim \tau_n$ are "signatures". Order doesn't matter. It can be weakened, do *injective* renaming (because we need the stability of disequality). These are not variables.
$\texttt{new}[\tau](c.m)$ is a class binder (binds the class c in m)

2.1 Statics

$$\frac{\Gamma \vdash_{\Sigma, c \sim \tau} v : \tau}{\Gamma \vdash_{\Sigma} \texttt{in}[c](v) : \texttt{clsfd}} \qquad \frac{\Gamma \vdash_{\Sigma, c \sim \tau} v : \texttt{clsfd} \qquad \Gamma, x : \tau \vdash_{\Sigma, c \sim \tau} m_1 \stackrel{.}{\sim} \tau \qquad \Gamma \vdash_{\Sigma, c \sim \tau} m_2 \stackrel{.}{\sim} \tau}{\Gamma \vdash_{\Sigma, c \sim \tau} \texttt{isin}[c](v; x.m_1; m_2) \stackrel{.}{\sim} \tau}$$

$$\frac{\Gamma \vdash_{\Sigma, c \sim \tau} m \stackrel{.}{\sim} \tau'}{\Gamma \vdash_{\Sigma} \texttt{new}[\tau](c.m) \stackrel{.}{\sim} \tau'} \qquad \frac{}{\Gamma \vdash_{\Sigma, c \sim \tau} \& c : \tau \ \texttt{cls}} \qquad \frac{\Gamma \vdash_{\Sigma} v : \tau \ \texttt{cls} \qquad \Gamma \vdash_{\Sigma, c \sim \tau} m \stackrel{.}{\sim} \tau'}{\Gamma \vdash_{\Sigma} \texttt{reveal}(v; c.m) \stackrel{.}{\sim} \tau'}$$

2.2 Dynamics

In a (scope-)free style,
states : $\nu\Sigma\{m\}$
transitions : $\nu\Sigma\{m\} \mapsto \nu\Sigma'\{m'\}$ with the invariant of $\Sigma' \supseteq \Sigma$
final states : $\nu\Sigma\{\texttt{ret}(v)\}$
When is a state statically OK? $\nu\Sigma\{m\}$ ok means $\vdash_{\Sigma} m \stackrel{.}{\sim} \tau$(for any τ). Compare with what we do for the static type($m \mapsto_{\Sigma} m'$).

$$\frac{}{\nu\Sigma\{\texttt{new}[\tau](c.m)\} \mapsto \nu(\Sigma, c \sim \tau)\{m\}}$$

$$\frac{c = c' (\Rightarrow \tau = \tau')}{\nu(\Sigma, c \sim \tau, c' \sim \tau')\{\texttt{isin}[c](\texttt{in}[c'](v'); x.m_1; m_2)\} \mapsto \nu(\Sigma, c \sim \tau, c' \sim \tau')\{[v'/x]m_1\}}$$

$$\frac{c \neq c'}{\nu(\Sigma, c \sim \tau, c' \sim \tau')\{\texttt{isin}[c](\texttt{in}[c'](v'); x.m_1; m_2)\} \mapsto \nu(\Sigma, c \sim \tau, c' \sim \tau')\{m_2\}}$$

$$\frac{}{\nu(\Sigma, c \sim \tau')\{\texttt{reveal}(\& c; c.m)\} \mapsto \nu(\Sigma, c \sim \tau)\{m\}}$$

$$\texttt{inref}(v, v') \triangleq \texttt{reveal}(v; c.m[c](v')) \stackrel{.}{\sim} \texttt{clsfd}$$

With all of this you can form a classified value without actually knowing what the classified value is.
Exercise: define $\texttt{isinref}(v; v'; x.m_1; m_2)$ (like isin but dynamically determined).

3 Capabilities

Possession of a reference (class reference) affords "full rights" to create and analyze classified values (according to that reference).

$$\tau \ \texttt{cls} \approx \langle \texttt{mk} \hookrightarrow \tau \to \texttt{clsfd}, \texttt{mk}^{-1} \hookrightarrow \texttt{clsfd} \to \tau \ \texttt{opt} \rangle$$

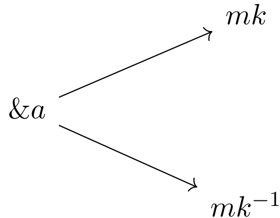
, where τ opt is just $\tau + 1$.

$$\& a \mapsto \langle \texttt{mk} \hookrightarrow \lambda[\tau](x.\texttt{in}[a](x)), \texttt{mk}^{-1} \hookrightarrow \lambda[\texttt{clsfd}](x.\texttt{isin}[a](x; y.\texttt{ret}(1 \cdot y); \texttt{ret}(2 \cdot \langle\rangle))) \rangle$$

What we want is dynamically "extensible" sum("$\tau_1 + \tau_2 + ... + \tau_n$"), and dynamically "new", aka, information flow or knowledge flow.
class is unguessable secret or perfect encryption(by $\alpha-$conversion).

capabilities express "integrity" and "confidentiality".
exceptions (i.e. the values) are "shared secrets".

1. The possessor of `mk` is held responsible for the integrity of the underlying data. The possessor, therefore, must ensure that invariant about the data hold.
2. The possessor of $\texttt{mk}^{-1}$ can compromise the confidentiality barrier and know that the associated invariant holds.



(To do some enforcement or analysis of security properties seems to involve some form of epistemic logic)

3.1 Relation to ML

In ML,

1. **exception** X **of** τ declares an exception X carrying values of type τ.
2. `raise`(e) raises an exception e : `exn` where `exn` is just `clsfd`.
3. e **handle** $z \Rightarrow e$ handles an exception: You don't need the name of the exception to handle it!

```
exception X of τ'
val X = ⟨raise[τ] ↪ exn ⇀ τ ,
         handle[τ] ↪ τ ⇀ (clsfd ⇀ τ) ⇀ τ⟩

raise X(v) ↪ bnd(comp(x.mk(v));
                 y.RAISE(y);
                 z.RAISE(t))

e handle X(x) => m' ↪
  bnd(comp(m); x. ret(x);
      y. bnd(comp(x.mk^-1(y)); x.m'; z.RAISE(z)))

exception X of τ ↪
  new[τ](c.⟨mk ↪ ...
           mk^-1 ↪ ...⟩)
```

In `raise` $X(v)$, X is of type $\tau \to$ `clsfd` and v is of type τ.

4 Things to look at

- Datatypes are abstract (*statically new*) recursive sums (static classification). Even if you write the same datatype declaration twice, you get two different types. Sometimes people call this a *generative data type*.

5 References

- Robert Harper. PFPL Supplement: Exceptions: Control and Data `https://www.cs.cmu.edu/~rwh/pfpl/supplements/exceptions.pdf`
- Robert Harper. Practical Foundations for Programming Languages. Chapter 29, 33.