Lecture Notes on Practical Foundations for Programming Languages

Part 4: Dynamic Classification

JIAWEN LIU, JÉRÉMY THIBAULT, and SHAOWEI ZHU

These are lecture notes taken during Robert Harper's course on foundations of programming languages at the Oregon Programming Language Summer School 2019. See also a PFPL supplement on exceptions at http://www.cs.cmu.edu/ rwh/pfpl/supplements/exceptions.pdf.

1 MOTIVATION

In this lecture, the goal is to build an exception control mechanism that:

- does not rely on any convention about the kind of data that is associated with an exception
- that ensures integrity and confidentiality of data: no third party should be able to intercept an exception intended for a particular handler

This is implemented by using dynamically created classes of data, and associating capabilities to them capabilities representing integrity and confidentiality of the associated data.

2 STATIC CLASSIFICATION

Dynamic classifications is to oppose to static classification, i.e., types built in the form of finite sums. Indeed, in a programming language equipped with static classification, a type is computed by solving a type equation:

$$\mathcal{U} \cong \left[\mathtt{num} \hookrightarrow \omega, \mathtt{nil} \hookrightarrow \mathbf{1}, \mathtt{cons} \hookrightarrow \mathcal{U} \times \mathcal{U}, \mathtt{fun} \hookrightarrow \mathcal{U} \to \mathcal{U}, \ldots\right]$$

where each summand is a class of values.

Dynamic dispatch can be seen as a way to work with the data typed as above. When we want to manipulate a value of this data type, we need to *unfold* it and then pattern-match on the type.

$$\mathtt{case}(u, \mathtt{num} \hookrightarrow \ldots; \ldots; \mathtt{cons} \hookrightarrow \langle x_1, x_2 \rangle)$$

The case analysis reveals further information about the type of the data that makes downstream manipulations possible.

Datatypes are *abstract* (statically "new": even you write the same implementation twice, they are two different types) recursive *sums* (static). Hence, this is an issue of using *generative* datatypes vs *concrete* datatypes.

3 DYNAMIC CLASSIFICATION

Now, we want a *dynamically extensible* sum: $\tau_1 + \tau_2 + \ldots$, generating new datatypes at runtime.

This idea of dynamically "new" connects closely to the notions of information-flow or knowledge-flow: a dynamic new class corresponds to an unguessable secret (or perfect encryption, by α-conversion), and one could control the flow of information by choosing who knows the secret.

We obtain abstract datatypes, with *induced capabilities* that express integrity and confidentiality, where

- the implementer is responsible for the integrity of data;

Authors' address: Jiawen Liu, jliu223@buffalo.edu; Jérémy Thibault, jeremy.thibault@inria.fr; Shaowei Zhu, shaoweiz@cs.princeton.edu.

- the client is responsible for confidentiality.

In particular, this may be used to implement exceptions: "shared secrets" between a raiser and a particular handler. In this case,

- the raiser is the implementer responsible for the integrity of the exception
- handlers other than the designated handler cannot decode the secret, thus are unable to handle the exception while being well-typed

Let us assume that there already exists an exception control-flow mechanism in our language. We will not focus on implementing the exception data itself.

4 IMPLEMENTATION OF THE EXCEPTION DATATYPE

Idea. We are going to associate with each exception a new class. An exception is raised by specifying its class; and an exception is handled by dispatching on the class.

4.1 Statics

$$\begin{array}{rll} \text{<Type>}\ \tau ::= \cdots \mid \underline{\texttt{clsfd}} & & \\ \text{<Value>}\ v ::= \cdots \mid \underline{\texttt{in}}\,[c]\,(v) & & : \underline{\texttt{clsfd}} \\ \text{<Comp>}\ m ::= \cdots \mid \underline{\texttt{isin}}\,[c]\,(v, x.m_1, m_2) & & \div\tau \\ \qquad \mid \underline{\texttt{new}}\,[\tau]\,(c.m) & & \end{array}$$

Here, c represents a class, used as an indexing scheme to distinguish between different exceptions. The computation $\underline{\texttt{isin}}\,[c]\,(v, x.m_1, m_2)$ type checks as τ if both $m_1 \div \tau$ and $m_2 \div \tau$, and $v : \underline{\texttt{clsfd}}$. Its semantics is that it checks if v is in c, and performs $[x/v]m_1$ in that case, or m_2 otherwise. Finally, $\underline{\texttt{new}}\,[\tau]\,(c.m)$ declares a new class c in m.

We also update our typing judgments to include an environment Σ of class binders associated with types.

$$\Gamma \vdash_\Sigma v : \tau$$

$$\Gamma \vdash_\Sigma m \div \tau$$

This environment Σ declares the active classes in the current scope. Note that, in our formalism, c is a class binder that is *not* a variable. In particular, the order in Σ does not matter, it is allowed to weaken Σ, and we can perform *injective* renaming, but not substitute (the renaming must be injective to preserve disequality).

The typing rule for "new" follows:

$$\frac{\Gamma \vdash_{\Sigma, a\sim\tau} m \div \tau'}{\Gamma \vdash_\Sigma \underline{\texttt{new}}\,[\tau]\,(a.m) \div \tau'}$$

4.2 Dynamics

We consider the following:

- states $\nu\Sigma\{m\}$: classes ν that could be used in computation m;
- transitions $\nu\Sigma\{m\} \mapsto \nu\Sigma'\{m'\}$: where the invariant $\Sigma \subseteq \Sigma'$ holds;
- final states $\nu\Sigma\{\underline{\texttt{ret}}\,(v)\}$

We introduce a static judgment, meaning that a state is well-formed:

$$\nu\Sigma\{m\}\mathrm{OK} \triangleq \cdot \vdash_\Sigma m \div \tau \text{ for any } \tau$$

The semantics of new is:

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

It allocates new classes of data, for "encryption" purposes.

The semantics of `in` and `isin` are:

$$\nu\Sigma, c \sim \tau, c' \sim \tau'\{\underline{\mathtt{isin}}\,[c]\,(\underline{\mathtt{in}}\,[c']\,(v'), x.m_1, m_2) \mapsto \begin{cases} [v'/x]m_1 \text{ if } c = c' \\ m_2 \text{ otherwise} \end{cases}.$$

The first case corresponds to the case where one learns the equality, and is able to perform the computation. In the second case, nothing at all is learnt.

4.3 Class references

Finally, we introduce the idea of class references.

$\tau ::= \cdots \mid \tau\ \underline{\mathtt{cls}}$	the type of class references
$v ::= \underline{\mathtt{cls}}\,[c]$	written as $\&c$
$m ::= \underline{\mathtt{reveal}}(v; c.m)$	

The additional typing rules are:

$$\frac{}{\Gamma \vdash_{\Sigma, c\ \tau} \&c : \tau\ \underline{\mathtt{cls}}} \qquad \frac{\Gamma \vdash_\Sigma v : \tau\ \underline{\mathtt{cls}} \qquad \Gamma \vdash_{\Sigma, c\sim\tau} m \div \tau'}{\Gamma \vdash_\Sigma \underline{\mathtt{reveal}}(v; c.m) \div \tau'}$$

Reveal has the following semantics:

$$\nu\Sigma, c \sim \tau\{\underline{\mathtt{reveal}}(\&c; c.m)\} \mapsto \nu\Sigma, c \sim \tau\{m\}$$

This corresponds to guessing by α-conversion.

We may now define two shortcuts, that behaves like "in" and "isin", but for class references:

$$\underline{\mathtt{inref}}(v; v') \triangleq \underline{\mathtt{reveal}}(v; c.\underline{\mathtt{in}}\,[c]\,(v')) \div \underline{\mathtt{clsfd}}$$
$$\underline{\mathtt{isinref}}(v; v'; x.m_1; m_2) \triangleq \underline{\mathtt{reveal}}(v; c.\underline{\mathtt{isin}}\,[c]\,(v', x.m_1, m_2)).$$

4.4 Possession of a class reference

Possession of a class reference affords *full rights* to create and analyze classified values. One may associate to a type $\tau\ \underline{\mathtt{cls}}$ the following tuple:

$$\tau\ \underline{\mathtt{cls}} \approx \left\{ \begin{array}{ll} \mathtt{mk}: & \tau \to \underline{\mathtt{clsfd}} \\ \mathtt{mk}^{-1}: & \underline{\mathtt{clsfd}} \to \tau\ \underline{\mathtt{opt}} \end{array} \right\}$$

and to the corresponding value $\&a$ the pair:

$$\&a \mapsto \left\{ \begin{array}{l} \mathtt{mk} \hookrightarrow \lambda[\tau](x : \underline{\mathtt{in}}\,[a]\,(x)) \\ \mathtt{mk}^{-1} \hookrightarrow \lambda[\underline{\mathtt{clsfd}}](x.\underline{\mathtt{isin}}\,[a]\,(x, y.\underline{\mathtt{ret}}\,(1.y)\,, \underline{\mathtt{ret}}\,(2.\ <>))) \end{array} \right\}$$

The following properties hold:

- the possessor of mk is held responsible for the integrity of the underlying value, and can enforce the value respects an invariant
- the possessor of $\mathtt{mk}^{-1}$ can compromise the confidentiality, and confirm that the invariant holds

5 AN APPLICATION IN ML

Let us introduce some concepts from the ML language.

- $\underline{\mathtt{exception}}\ X\ \underline{\mathtt{of}}\ \tau$ declares an exception X carrying values of type τ
- $\underline{\mathtt{raise}}\ e$ where $e : \underline{\mathtt{exn}}$ (that is, $\underline{\mathtt{clsfd}}$) raises an exception
- $e\ \underline{\mathtt{handle}}\ X(x)$ declares an handler. However, one might as well write z instead of $X(x)$, because one doesn't need to have access to the name of the exception to handle it.

So what does $\underline{\mathtt{exception}}\ X\ \underline{\mathtt{of}}\ \tau$ actually mean?

$$\underline{\mathtt{val}}\ X = \left\{ \begin{array}{ll} \underline{\mathtt{raise}}[\tau] & \hookrightarrow \underline{\mathtt{clsfd}} \to \tau \\ \underline{\mathtt{handle}}[\tau] & \hookrightarrow \tau \to (\underline{\mathtt{clsfd}} \to \tau) \to \tau \end{array} \right\}$$

The handler takes two parameters: the expression to evaluate, and a function to call if an exception is raised.

In ML, $\underline{\mathtt{raise}}$ has the following meaning, where $X : \tau \to \underline{\mathtt{clsfd}}$ and $x : \tau$:

$$\underline{\mathtt{raise}}\ X(v) \hookrightarrow \underline{\mathtt{bnd}}(\underline{\mathtt{comp}}(X.\mathtt{mk}(v)), y.\mathtt{RAISE}(y), z.\mathtt{RAISE}(z))$$

This computes the content of the exception, classify it, and then calls the system implementation for raising an exception. The second RAISE is used if the inner computation raises an exception itself.

Similarly, $\underline{\mathtt{handle}}$ has the following meaning:

$$m\ \underline{\mathtt{handle}}\ X(x) \Rightarrow m' \hookrightarrow \underline{\mathtt{bnd}}(\underline{\mathtt{comp}}(m), x.\underline{\mathtt{ret}}\,(x)\,, y.\underline{\mathtt{bnd}}(\underline{\mathtt{comp}}(x.\mathtt{mk}^{-1}(y)), x.m', z.\mathtt{RAISE}(z)))$$

If the execution of m succeeds, the value is immediately returned. Otherwise, the exception is decoded ($\mathtt{mk}^{-1}$), then handled by m'. Again, one need to handle the case of an inner exception.

Finally, it is now possible to see exceptions as dynamically allocating a new class:

$$\underline{\mathtt{exception}}\ X\ \underline{\mathtt{of}}\ \tau \hookrightarrow \underline{\mathtt{new}}\,[\tau]\,(c.\langle \mathtt{mk} \hookrightarrow \ldots, \mathtt{mk}^{-1} \hookrightarrow \ldots \rangle)$$