Recitation 14:
Dynamic Classification

15-312: Foundations of Programming Languages

Jeanne Luning Prak

April 25th, 2018

1 Motivation

In Recitation 9, we briefly talked about how all exception values must have the same type, since the handler cannot know where a raised exception comes from. We went through several possible types for exception values, including nat (error codes) and string (error messages). However, each has its drawbacks. Error numbers must be agreed upon ahead of time to be useful, and error messages are useful to a human debugging, but not an exception handler in your code. We also considered a sum type with every type we plan to use as an exception for our program, but this is very anti-modular.

A better approach is to have a single type that can be *extended* with new *classes* for different types. Symbols[1] generated at runtime are used to tag values of various types, and all tagged values have type `clsfd` (classified)[2]. This is, in fact, what Standard ML does for its exception type: `exn` can be extended with new dynamic classes using the `exception` keyword.

```
exception Message of string
```

creates a new symbol `Message` to tag values of type string to create an value of type `exn`. For this reason, we say that the `exn` type is an extensible type.

In this recitation, we'll formalize this `clsfd` type

2 Clsfd

We add three new operators that introduce and eliminate values of type `clsfd`.

Sort	Abstract Syntax	Concrete Syntax
Typ τ ::=	`clsfd`	`clsfd`
Exp e ::=	`in[a](e)`	$a \cdot e$
	$\texttt{isin}[\mathsf{a}](\mathsf{e}; \mathsf{x}.\mathsf{e}_1; \mathsf{e}_2)$	$\texttt{match } \mathsf{e} \texttt{ as } \mathsf{a} \cdot \mathsf{x} \hookrightarrow \mathsf{e}_1 \texttt{ ow} \hookrightarrow \mathsf{e}_2$

[1]To get a good understanding of symbols, take a look at Chapter 31 of PFPL.

[2]You may remember `clsfd` from Concurrent Algol.

3 Statics

To create a value of type `clsfd`, we use a symbol a associated with type τ to classify a value e of type τ in `in[a](e)`. We can also check if a value of type `clsfd` is classified with a particular symbol using `isin`, binding the value that was classified to x and evaluating e_1 if it is, and evaluating e_2 otherwise.

To formally define the statics, we use a symbol signature Σ which contains the symbols in scope and their associated types. As we saw in Modernized Algol, this is distinct from a variable context, as symbols are not given meaning by substitution, but instead exist as their own atomic units.

$$\frac{\Gamma \vdash_{\Sigma, a \sim \tau} e : \tau}{\Gamma \vdash_{\Sigma, a \sim \tau} \texttt{in[a](e)} : \texttt{clsfd}} \qquad \frac{\Gamma \vdash_{\Sigma, a \sim \tau} e : \texttt{clsfd} \quad \Gamma, x : \tau \vdash_{\Sigma, a \sim \tau} e_1 : \tau' \quad \Gamma \vdash_{\Sigma, a \sim \tau} e_2 : \tau'}{\Gamma \vdash_{\Sigma, a \sim \tau} \texttt{isin}[\mathsf{a}](\mathsf{e}; \mathtt{x}.\mathsf{e}_1; \mathsf{e}_2) : \tau'}$$

4 Dynamics

We use $\nu\ \Sigma$ to define the dynamics, where Σ is the symbols currently in scope[3]. This gives us a *scope-free* dynamics in which a symbol, once created, remains in scope for every future expression. We never remove symbols from Σ; we only add them.

$$\frac{e\ \texttt{val}_\Sigma}{\texttt{in}[\mathsf{a}](\mathsf{e})\ \texttt{val}_\Sigma} \qquad \frac{\nu\ \Sigma\{e\} \longmapsto \nu\ \Sigma'\{e'\}}{\nu\ \Sigma\{\texttt{in}[\mathsf{a}](\mathsf{e})\} \longmapsto \nu\ \Sigma'\{\texttt{in}[\mathsf{a}](\mathsf{e}')\}}$$

$$\frac{e\ \texttt{val}_\Sigma}{\nu\ \Sigma\{\texttt{isin}[\mathsf{a}](\texttt{in}[\mathsf{a}](\mathsf{e}); \mathtt{x}.\mathsf{e}_1; \mathsf{e}_2)\} \longmapsto \nu\ \Sigma\{[e/x]e_1\}}$$

$$\frac{e'\ \texttt{val}_\Sigma \quad (a \neq a')}{\nu\ \Sigma\{\texttt{isin}[\mathsf{a}](\texttt{in}[\mathsf{a}'](\mathsf{e}'); \mathtt{x}.\mathsf{e}_1; \mathsf{e}_2)\} \longmapsto \nu\ \Sigma\{e_2\}}$$

$$\frac{\nu\ \Sigma\{e\} \longmapsto \nu\ \Sigma'\{e'\}}{\nu\ \Sigma\{\texttt{isin}[\mathsf{a}](\mathsf{e}; \mathtt{x}.\mathsf{e}_1; \mathsf{e}_2)\} \longmapsto \nu\ \Sigma\{\texttt{isin}[\mathsf{a}](\mathsf{e}'; \mathtt{x}.\mathsf{e}_1; \mathsf{e}_2)\}}$$

It's worth noting that these rules look somewhat similar to the rules for `in` and `case` for sum types. However, for a sum type, all of the labels are known statically and we can check if the case is exhaustive. For a value of type `clsfd`, all of the symbols that it could be tagged with are not known, and, in fact, cannot be known, since more can be dynamically generated. This means that only an expression that has a particular symbol in scope can match on it. This gives dynamically classified values a sort of confidentiality[4] and integrity, as only someone with the symbol can tag a value with it and only someone with the symbol can retrieve the value tagged with that symbol.

[3] The ν is just a symbol; it has no meaning.
[4] Which gives us an excellent pun on the word "classified"

5 Examples

5.1 Exceptions in SML

Let's look at an example of exceptions in Standard ML.

```
exception FoundZero of int

fun foo (x : int) = if x = 0 then raise FoundZero 0 else x

val _ = foo 0 handle FoundZero x => x
```

We can translate this into `XPCF`[5] with `clsfd`.

```
new{int}(foundZero.
    try(
       (fn (x : int) ifz(x; raise(in[foundZero](z)); _.x))(z);
        ex.isin[foundZero](ex; n.n; raise(ex))
    )
)
```

Notice that a handle translates to a `try` and an `isin` where the `ow` case of the `isin` re-raises the exception. This is consistent with the behavior of exceptions in Standard ML: if an exception is not handled by a handler, it remains raised.

5.2 Channels in CA

We can have a form of "selective" communication using broadcast communication if we broadcast a `clsfd` value with a channel symbol, so that only processes with that channel in scope can match against it. Say Alice wants to send the number 8128 to Bob without Eve knowing what was sent. Alice can do this by broadcasting along a channel that Eve does not have access to:

```
new[nat](b.
    run(msg ← acc;                        (* Eve *)
           match msg with
              b n => ret n
              ow  => _ ← emit(msg);
                     ret 0)
    ⊗ new[nat](a.run(emit(a(8128))     (* Alice *)
               ⊗ run(msg ← acc;        (* Bob *)
                     match msg with
                        a n => ret n
                        ow  => _ ← emit(msg);
                               ret 0)))
```

We use the concrete syntax here for readability. Notice that channel a is not in scope for the first process, so it would have no hope of decoding the message (only of stopping its transmission).

[5] If you remember, this PCF with exceptions