Harvard School of Engineering and Applied Sciences — CS 152: Programming Languages

# Dynamic types

Lecture 19 Tuesday, April 9, 2013

## 1 Error-propagating semantics

For the last few weeks, we have been studying type systems. As we have seen, types can be useful for reasoning about a program's execution (before ever actually executing the program!) and also for restricting the use of values or computations (for example, using existential types to provide encapsulation).

However, many currently popular languages (including JavaScript, Perl, PHP, Python, Ruby, and Scheme) do not have static type systems. None-the-less, during execution these languages manipulate values of different types, but when a value is used in an inappropriate way, the program does not get stuck, but instead causes an error at runtime.

To model these dynamic type errors, let's extend an (untyped) lambda calculus with a special Err value.

$$\begin{aligned} e &::= x \mid \lambda x.\, e \mid e_1\ e_2 \mid n \mid e_1 + e_2 \mid \mathsf{Err} \\ v &::= n \mid \lambda x.\, e \mid \mathsf{Err} \end{aligned}$$

Note that Err is not part of the "surface syntax", i.e., the source program will not contain Err, but instead this special value will only arise during execution. (This is similar to locations $\ell$, which are created by allocation, but do not appear in source programs.)

The intention is that if we dynamically encounter a type error (e.g., try to add two functions together, or apply an integer as if it were a function), we will produce the Err value.

$$E ::= E\ e \mid v\ E \mid E + e \mid v + E$$

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad \frac{}{(\lambda x.\, e)\ v \longrightarrow e\{v/x\}}\ v \neq \mathsf{Err} \qquad \frac{}{n_1 + n_2 \longrightarrow n}\ n = n_1 + n_2$$

$$\frac{}{v_1\ v_2 \longrightarrow \mathsf{Err}}\ v_1 \neq \lambda x.\, e \qquad \frac{}{v_1 + v_2 \longrightarrow \mathsf{Err}}\ v_1 \text{ or } v_2 \text{ not an integer}$$

We also need some rules to propagate errors. For example, if the argument to a function is an error.

$$\frac{}{(\lambda x.\, e)\ \mathsf{Err} \longrightarrow \mathsf{Err}}$$

Let's see an example of a program executing.

$$\begin{aligned} & 42 + ((\lambda f.\, \lambda n.\, f\ (n + 3))\ (\lambda x.\, x)\ (\lambda x.\, x)) \\ \longrightarrow & 42 + ((\lambda n.\, (\lambda x.\, x)\ (n + 3))\ (\lambda x.\, x)) \\ \longrightarrow & 42 + ((\lambda x.\, x)\ ((\lambda x.\, x) + 3)) \\ \longrightarrow & 42 + ((\lambda x.\, x)\ \mathsf{Err}) \\ \longrightarrow & 42 + \mathsf{Err} \\ \longrightarrow & \mathsf{Err} \end{aligned}$$

Note that once an error has occurred in a subexpression, the error will propagate up to the top level.

In our simple calculus, it is easy for us to determine syntactically whether a particular value is an integer or a function, and thus to figure out whether it is being used appropriately. In an implementation, however, we may need to ensure that at run time we have some way of distinguishing values of different types. For example, we wouldn't be able to implement this error-propagating semantics if both integers and functions were represented using 32 bits with no way to distinguish them (e.g., 32-bit signed integers, and 32-bit pointers to function bodies).

Given that we need this run-time information to distinguish values of different types, we could provide primitives to allow the programmer to query the type of a value. This would allow a careful or paranoid programmer to avoid dynamic type errors. We extend the language with booleans, conditionals, and primitives to check whether a value is an integer or a function.

$$\begin{aligned}
e &::= \cdots \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \mathsf{is\_int?}\ e \mid \mathsf{is\_fun?}\ e \mid \mathsf{is\_bool?}\ e \\
v &::= \cdots \mid \mathsf{true} \mid \mathsf{false} \\
E &::= \cdots \mid \mathsf{if}\ E\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \mathsf{is\_int?}\ E \mid \mathsf{is\_fun?}\ E \mid \mathsf{is\_bool?}\ E
\end{aligned}$$

$$\frac{}{\mathsf{if\ true\ then}\ e_2\ \mathsf{else}\ e_3 \longrightarrow e_2} \qquad \frac{}{\mathsf{if\ false\ then}\ e_2\ \mathsf{else}\ e_3 \longrightarrow e_3}$$

$$\frac{}{\mathsf{if}\ v\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \longrightarrow \mathsf{Err}}\ v \neq \mathsf{true}\ \text{and}\ v \neq \mathsf{false}$$

$$\frac{}{\mathsf{is\_int?}\ n \longrightarrow \mathsf{true}} \qquad \frac{}{\mathsf{is\_int?}\ v \longrightarrow \mathsf{false}}\ v\ \text{not an integer and}\ v \neq \mathsf{Err} \qquad \frac{}{\mathsf{is\_fun?}\ \lambda x.\, e \longrightarrow \mathsf{true}}$$

$$\frac{}{\mathsf{is\_fun?}\ v \longrightarrow \mathsf{false}}\ v \neq \lambda x.\, e\ \text{and}\ v \neq \mathsf{Err} \qquad \frac{}{\mathsf{is\_bool?}\ v \longrightarrow \mathsf{true}}\ v = \mathsf{true}\ \text{or}\ v = \mathsf{false}$$

$$\frac{}{\mathsf{is\_bool?}\ v \longrightarrow \mathsf{false}}\ v \notin \{\mathsf{true}, \mathsf{false}, \mathsf{Err}\}$$

Again, we need some additional rules to propagate errors.

$$\frac{}{\mathsf{is\_int?\ Err} \longrightarrow \mathsf{Err}} \quad \frac{}{\mathsf{is\_fun?\ Err} \longrightarrow \mathsf{Err}} \quad \frac{}{\mathsf{is\_bool?\ Err} \longrightarrow \mathsf{Err}}$$

## 2 Exception handling

As mentioned above, once an error has occurred in a subexpression, it propagates up to the top level. However, if a programmer knows how to handle an error, then we should perhaps allow the programmer to "catch" or handle the error. Indeed, let's give the programmer an explicit mechanism to raise an error. This is similar to an exception mechanism, where exceptions can be raised (also known as "throwing" an exception), and then caught by handlers. We could extend the language to have different kinds of errors, or exceptions, that can be raised, and extend our handler mechanism to selectively catch errors. Let's not go quite that far, but we will add values to errors.

$$\begin{aligned}
e &::= \cdots \mid \mathsf{try}\ e_1\ \mathsf{catch}\ x.\ e_2 \mid \mathsf{raise}\ e \\
v &::= \cdots \mid \mathsf{Err}\ v \\
E &::= \cdots \mid \mathsf{try}\ E\ \mathsf{catch}\ x.\ e_2 \mid \mathsf{raise}\ E
\end{aligned}$$

The raise primitive raises an error, while try $e_1$ catch $x.\ e_2$ will evaluation $e_1$, and evaluate $e_2$ with $x$ bound to value $v$ only if $e_1$ evaluates to Err $v$.

$$\frac{}{\mathsf{raise}\ v \longrightarrow \mathsf{Err}\ v}\ v \neq \mathsf{Err}\ v' \qquad \frac{}{\mathsf{raise}\ (\mathsf{Err}\ v) \longrightarrow \mathsf{Err}\ v}$$

$$\frac{}{\mathsf{try\ Err}\ v\ \mathsf{catch}\ x.\ e_2 \longrightarrow e_2\{v/x\}} \qquad \frac{}{\mathsf{try}\ v\ \mathsf{catch}\ x.\ e_2 \longrightarrow v}\ v \neq \mathsf{Err}\ v'$$

We give new rules for creating errors, so that when evaluation raises an error, it has a value associated with it. Here we use the integer zero to indicate that a non-function value was applied, integer 1 to indicate that a non-integer value was used as an operand for addition, and integer 2 to indicate that a non-boolean value was used as the test for a conditional. Of course, in a more expressive language, we may use strings to describe the errors.

$$\frac{}{v_1\ v_2 \longrightarrow \mathsf{Err}\ 0}\ v_1 \neq \lambda x.\, e \qquad \frac{}{v_1 + v_2 \longrightarrow \mathsf{Err}\ 1}\ v_1 \text{ or } v_2 \text{ not an integer}$$

$$\frac{}{\mathsf{if}\ v\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \longrightarrow \mathsf{Err}\ 2}\ v \neq \mathsf{true} \text{ and } v \neq \mathsf{false}$$

(We would, of course, need to also modify the previous rules we presented in Section 1 to account for the fact that Err now has a value associated with it. This is straightforward, and we omit the details.)

Consider the following program:

$$\begin{array}{l}\mathsf{let}\ \mathit{foo} = \lambda x.\, \mathsf{if}\ \mathsf{is\_int?}\ x\ \mathsf{then}\ x + 7\ \mathsf{else}\ \mathsf{raise}\ (x+1) \\ \ \mathsf{in}\ \mathit{foo}\ (\lambda y.\, y)\end{array}$$

What happens when we execute it?

## 3 Contracts

We introduced primitive operations to distinguish integers, booleans, and functions. A defensive programmer may insert code to check that values have the expected type. But what happens if we need to use third-party code (i.e., code that is not under our control)? We will not be able to insert checks into that code.

This means that we may not be able to quickly detect type errors dynamically: the actual type error could occur significantly after a value was produced that failed to meet the programmer's expectations.

Consider the following example.

$$\begin{array}{l}\mathsf{let}\ \mathit{double} = \lambda f.\, f\ (f\ 0)\ \mathsf{in} \\ \mathsf{let}\ \mathit{pos} = \lambda i.\, \mathsf{if}\ i < 0\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \mathsf{true}\ \mathsf{in} \\ \mathit{double}\ \mathit{pos}\end{array}$$

Function *double* takes in a function $f$, applies it to zero, and applies $f$ again to the result. Clearly, *double* is expecting $f$ to be a function that accepts integers, and returns integers. Function *pos*, however, takes an integer and returns a boolean. Let's see what happens when we run the program.

$$\begin{array}{rl} & \mathit{double}\ \mathit{pos} \\ \longrightarrow & \mathit{pos}\ (\mathit{pos}\ 0) \\ \longrightarrow^* & \mathit{pos}\ \mathsf{true} \\ \longrightarrow & \mathsf{if}\ \mathsf{true} < 0\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \mathsf{true} \\ \longrightarrow & \mathsf{Err}\ 2\end{array}$$

When we run the program, we get a runtime error in the (second) execution of *pos*. Suppose this were a larger program, and think about what would be required to debug this. First, we would need to realize that the error occured in the execution of *pos*, due to the argument to *pos* being inappropriate. Then we need to figure out that the call that passed the wrong argument occurred in *double*. Then we need to figure out that the argument was the result of calling $f\ 0$, then we need to figure out that $f$ was instantiated with *pos*. Then, we need to look into the code of *pos* to see why the result is not an integer. **phew**.

Even in this simple situation, the debugging process requires quite some work, as the actual type error occurs in a place not directly related to the real source of the error.

However, our debugging process revealed that the programmer (i.e., the producer of function $double$) made some assumptions about how $f$ is supposed to work. If the programmer could state those assumptions explicitly, then perhaps we would be able to produce an error sooner, and closer to the real source of the issue. Similarly, the developer of $pos$ made some assumptions too, and stating those assumptions is critical for $pos$ to avoid being abused. As mentioned, defensive programmers could insert defensive checks into their code, but this would lead to code clutter, obscuring and polluting the code. Also, it is not necessarily easy to write all the checks correctly.

We introduce a new mechanism to help us ensure that function values act appropriately on the arguments they accept and the results they produce. This calculus is a simplification of *higher order contracts*[1], a language feature that has not yet found its way into popular scripting languages, but which allows for dynamic type checking, even in the presence of first-order functions.

A *flat contract* is simply a function that accepts a value, and returns true if the value meets the contract (i.e., is good, or acceptable), and returns false otherwise. A *monitor* $\mathsf{monitor}(e_1, e_2)$ combines a computation $e_1$ with a contract $e_2$ and will evaluate $e_1$ and $e_2$ to values $v_1$ and $v_2$, and then apply the contract $v_2$ to the value $v_1$. If the contract says that $v_1$ is acceptable, then execution continues using $v_1$; otherwise a run time error is raised.

Flat contracts allow us to check that values are integers, booleans, etc.[2] However, flat contracts are't suitable for ensuring that functions accept and produce values of appropriate types.

We introduce a *function contract* $e_1 \longmapsto e_2$, which we will use to monitor evaluation of functions. Expressions $e_1$ and $e_2$ are contracts, and when we apply a function to an argument $v$, we will use contract $e_1$ to make sure that $v$ is an appropriate argument, and, if the evaluation of the function application terminates, then we will use contract $e_2$ to check that the result is appropraite.

The following describes the syntax and semantics for extending our language with function contracts and monitors.

$$\begin{aligned}
e &::= \cdots \mid e_1 \longmapsto e_2 \mid \mathsf{monitor}(e_1, e_2)\\
v &::= \cdots \mid v_1 \longmapsto v_2 \mid \mathsf{monitor}(v, v_1 \longmapsto v_2)\\
E &::= \cdots \mid E \longmapsto v \mid v \longmapsto E \mid \mathsf{monitor}(e, E) \mid \mathsf{monitor}(E, v)
\end{aligned}$$

$$\frac{}{(\mathsf{monitor}(v, v_1 \longmapsto v_2))\ v' \longrightarrow \mathsf{monitor}((v\ (\mathsf{monitor}(v', v_1))), v_2)}$$

$$\frac{}{\mathsf{monitor}(v, v') \longrightarrow \mathsf{if}\ v'\ v\ \mathsf{then}\ v\ \mathsf{else\ raise}\ 3}\ v' \neq v_1 \longmapsto v_2$$

Intuitively, when we have a function application $\lambda x.\, e$ that is being monitored by contract $v_1 \longmapsto v_2$, we first make sure that argument $v$ passes contract $v_1$ (using $\mathsf{monitor}(v_1, v)$), apply the function to the result, and make sure that the result of the function application will have contract $v_2$ called on it ($\mathsf{monitor}((v\ (v_1\ v')), v_2)$).

We also need some rules to propagate error values.

$$\frac{}{(\mathsf{Err}\ v \longmapsto e) \longrightarrow \mathsf{Err}\ v} \qquad \frac{}{(e \longmapsto \mathsf{Err}\ v) \longrightarrow \mathsf{Err}\ v} \qquad \frac{}{\mathsf{monitor}(\mathsf{Err}\ v, e) \longrightarrow \mathsf{Err}\ v}$$

$$\frac{}{\mathsf{monitor}(e, \mathsf{Err}\ v) \longrightarrow \mathsf{Err}\ v}$$

Let's take a look at our $double$ example again, this time using a function contract to ensure that $double$ takes as input, a function from integers to integers, and returns an integer. We also put a function contract on $pos$ to show that it is a function from integers to booleans. That is, we are making the specification of

[1]For more information, see *Contracts for Higher-Order Functions* by Findler and Felleisen, in *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, 2002.

[2]In fact, contracts can go beyond simple static type checking, and check arbitrary properties, for example, that an integer is even. For the purposes of this class, we'll just consider type-checking-like properties.

both $double$ and $pos$ explicit in the code, via contracts, and are checking these specifications as the program executes.

$$\begin{aligned}
&\text{let } double = \mathsf{monitor}(\lambda f.\, f\ (f\ 0), (\mathsf{is\_int?} \longmapsto \mathsf{is\_int?}) \longmapsto \mathsf{is\_int?}) \text{ in}\\
&\text{let } pos = \mathsf{monitor}(\lambda i.\, \mathsf{if}\ i < 0\ \mathsf{then\ false\ else\ true}, \mathsf{is\_int?} \longmapsto \mathsf{is\_bool?}) \text{ in}\\
&double\ pos
\end{aligned}$$

Note that the contract for $double$ is $(\mathsf{is\_int?} \longmapsto \mathsf{is\_int?}) \longmapsto \mathsf{is\_int?}$, indicating that it takes as an argument a value satisfying function contract $\mathsf{is\_int?} \longmapsto \mathsf{is\_int?}$ and will return a value satisfying the flat contract $\mathsf{is\_int?}$. Similarly, the contract for $pos$ indicates that it takes an integer as an argument and returns a boolean.

Let's consider the execution of this program.

$$\begin{aligned}
&double\ pos\\
=&(\mathsf{monitor}(\lambda f.\, f\ (f\ 0), (\mathsf{is\_int?} \longmapsto \mathsf{is\_int?}) \longmapsto \mathsf{is\_int?}))\ pos\\
\longrightarrow&\mathsf{monitor}((\lambda f.\, f\ (f\ 0))\ (\mathsf{monitor}(pos, \mathsf{is\_int?} \longmapsto \mathsf{is\_int?})), \mathsf{is\_int?})\\
=&\mathsf{monitor}((\lambda f.\, f\ (f\ 0))\\
&\quad(\mathsf{monitor}(\mathsf{monitor}(\lambda i.\, \mathsf{if}\ i < 0\ \mathsf{then\ false\ else\ true}, \mathsf{is\_int?} \longmapsto \mathsf{is\_bool?}), \mathsf{is\_int?} \longmapsto \mathsf{is\_int?})), \mathsf{is\_int?})\\
\longrightarrow&\mathsf{monitor}((M\ (M\ 0)), \mathsf{is\_int?})
\end{aligned}$$

where $M = \mathsf{monitor}(\mathsf{monitor}(\lambda i.\, \mathsf{if}\ i < 0\ \mathsf{then\ false\ else\ true}, \mathsf{is\_int?} \longmapsto \mathsf{is\_bool?}), \mathsf{is\_int?} \longmapsto \mathsf{is\_int?})$.

Let's consider the evaluation of $M\ 0$.

$$\begin{aligned}
&M\ 0\\
=&(\mathsf{monitor}(N, \mathsf{is\_int?} \longmapsto \mathsf{is\_int?}))\ 0
\end{aligned}$$

where $N = \mathsf{monitor}(\lambda i.\, \mathsf{if}\ i < 0\ \mathsf{then\ false\ else\ true}, \mathsf{is\_int?} \longmapsto \mathsf{is\_bool?})$

$$\begin{aligned}
\longrightarrow&\mathsf{monitor}(N\ \mathsf{monitor}(0, \mathsf{is\_int?}), \mathsf{is\_int?})\\
\longrightarrow^*&\mathsf{monitor}(N\ 0, \mathsf{is\_int?})\\
\longrightarrow&\mathsf{monitor}(\mathsf{monitor}((\lambda i.\, \mathsf{if}\ i < 0\ \mathsf{then\ false\ else\ true})\ \mathsf{monitor}(0, \mathsf{is\_int?}), \mathsf{is\_bool?}), \mathsf{is\_int?})\\
\longrightarrow^*&\mathsf{monitor}(\mathsf{monitor}((\lambda i.\, \mathsf{if}\ i < 0\ \mathsf{then\ false\ else\ true})\ 0, \mathsf{is\_bool?}), \mathsf{is\_int?})\\
\longrightarrow&\mathsf{monitor}(\mathsf{monitor}((\mathsf{if}\ 0 < 0\ \mathsf{then\ false\ else\ true}), \mathsf{is\_bool?}), \mathsf{is\_int?})\\
\longrightarrow&\mathsf{monitor}(\mathsf{monitor}(\mathsf{true}, \mathsf{is\_bool?}), \mathsf{is\_int?})\\
\longrightarrow^*&\mathsf{monitor}(\mathsf{true}, \mathsf{is\_int?})\\
\longrightarrow^*&\mathsf{Err}\ 3
\end{aligned}$$

So what happened here? Again, we got a run time error as a result of executing this program. But note that the error happened sooner than in the original execution: it happened at the end of the first execution of $pos$. Moreover, the contract that raised the error was the contract for $double$, specifically the part of the contract that stated that argument $f$ should be a function from integers to integers. Indeed, that was the part of the specification that was violated: the argument $f$, when applied to an integer value, did not return an integer.

### 3.1 Blame

In the example above, the function contract correctly detected that $pos\ 0$ does not evaluate to an integer. But who was to blame for this error? That is, did the developer of function $pos$ implement that function incorrectly? Or was it the case that function $double$ was using its argument $f$ incorrectly? In essence, how do we decide which piece of code is to blame for the error?

Knowing who to blame can be useful, not just for pointing fingers, but also for debugging: where is the error in the code? This can be difficult to figure out in a higher order setting, because if a function is used incorrectly (i.e., is given an inappropriate argument) the actual error may occur quite a bit later in the execution.

We can extend the calculus above to keep track of blame, and blame the correct entity. Intuitively, all a monitor needs to do is keep track of two entities: a label $p$ representing the provider of the code that it is monitoring (i.e., who is producing the value), and a label $n$ representing the context in which the monitor occurs. The idea is that if a function is being monitored, then $n$ may provide arguments to the function. We can think of these labels $p$ and $n$ as being module names, or providers of code. In our example above, there are three entities involved, and we would perhaps have three different labels: one label for code from the function $double$, one label for code from the function $pos$, and one label for the "main" code that applied $double$ to $pos$.

The rules for the monitor keep track of who is to blame. The rules for these blame-tracking monitors are as follows:

$$\frac{}{(\mathsf{monitor}(v, v_1 \longmapsto v_2, p, n))\ v' \longrightarrow \mathsf{monitor}((v\ (\mathsf{monitor}(v', v_1, n, p))), v_2, p, n)}$$

$$\frac{}{\mathsf{monitor}(v, v', p, n) \longrightarrow \mathsf{if}\ v'\ v\ \mathsf{then}\ v\ \mathsf{else\ raise}\ \text{“Blame } p\text{”}}\ v' \neq v_1 \longmapsto v_2$$

The key things to note is that for a monitored function application $(\mathsf{monitor}(v, v_1 \longmapsto v_2, p, n))\ v'$, if $v'$ does not satisfy contract $v_1$, then $n$ will be blamed, i.e., the provider of value $v'$. By contrast, if the result of the function application doesn't satisfy contract $v_2$, then $p$ will be blamed.

In our example above, the entity to blame is the one that wrote the code that applies $double$ to $pos$. Both functions $double$ and $pos$ meet their specification. The problem with our example program is that $double$ is given an argument that doesn't satisfy the contract for the argument to $double$.

As a final note, where do the labels come from? It is not the programmer's job to identify the labels for the monitor. Rather, the language compiler and runtime is resposible for selecting the labels. The positive party $p$ for a monitor can be identified immediately: it is the label of the code where the monitor is defined. Identification of the negative party $n$ for a monitor is delayed until a value is used. That is, it is when a monitored function is applied, that the negative label is set to be the provider of the argument. For function $double$, the provider of the argument is "main" code, i.e., the code that applies $double$ to $pos$.