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

Dynamic types

Lecture 19 — Thursday, April 9, 2015

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 ℓ, 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.

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

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

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

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

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

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

Again, we need some additional rules to propagate errors.

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

As a concluding remark in this section, notice that instead of duplicating almost all of our inference rules to propagate errors one evaluation context frame per evaluation step, a single rule would be sufficient to immediately abort evaluation with an error:

$$\frac{e \longrightarrow \mathsf{Err}}{E[e] \longrightarrow \mathsf{Err}}$$

When an error occurs, this rule discards the remainder of the program (i.e., the context E) and returns the error immediately as the result of the evaluation of the program. On one hand, this alternative formulation simplifies the semantics of our small calculus. On the other hand, it complicates the extension of the calculus with exception handlers (which we discuss next). Thus for the remainder of the lecture we stick to rules that propagate errors one evaluation context frame per evaluation step.

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, and 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.

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

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

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.

$$\frac{}{v_1\ v_2 \longrightarrow \mathsf{Err}\ 0}\ v_1 \neq \lambda x.\ e \text{ and } v_1 \neq \mathsf{Err}\ v \qquad \frac{}{(\mathsf{Err}\ v_1)\ v_2 \longrightarrow \mathsf{Err}\ v_1}$$

$$\frac{}{v_1 + v_2 \longrightarrow \mathsf{Err}\ 1}\ v_1 \text{ or } v_2 \text{ not an integer and } v_1 \neq \mathsf{Err}\ v \text{ and } v_2 \neq \mathsf{Err}\ v' \qquad \frac{}{(\mathsf{Err}\ v_1) + v_2 \longrightarrow \mathsf{Err}\ v_1}$$

$$\frac{}{n_1 + \mathsf{Err}\ v \longrightarrow \mathsf{Err}\ v} \qquad \frac{}{\mathsf{if}\ v\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \longrightarrow \mathsf{Err}\ 2}\ v \neq \mathsf{true} \text{ and } v \neq \mathsf{false} \text{ and } v \neq \mathsf{Err}\ v'$$

$$\frac{}{\mathsf{if}\ \mathsf{Err}\ v\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \longrightarrow \mathsf{Err}\ v}$$

(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:

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

What happens when we execute it?

3 Contracts

We introduced primitive operations to distinguish integers, booleans, and functions. These operations permit a defensive programmer to insert checks that ensure that values are of the expected type at a given point of the program execution.

However, introducing these checks into a program is not always easy or possible. First, programmers must work hard to insert checks at the right places. Second, they must work even harder to maintain checking code as programs evolve. Third, missing checks may lead to type errors that go undetected until a latter step in the execution of a program. As a result, programmers may end up looking at the wrong place for a bug after a check raises an error. Fourth, for programs that expect functions as arguments, it is not possible for programmers to modify the functions-arguments and inject checks in them. To partially cope with this latter problem, programmers must insert even more subtle checks to their code to make sure that the functions their code consumes behave as expected (consider how the process of inserting checks becomes more and more complicated as programs consume functions that consume functions that consume functions etc.).

The following example demonstrates some of the issues we discuss above.

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

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.

$$\begin{aligned}
&\mathit{double}\ \mathit{pos}\\
\longrightarrow\ &\mathit{pos}\ (\mathit{pos}\ 0)\\
\longrightarrow^*\ &\mathit{pos}\ \mathsf{true}\\
\longrightarrow\ &\mathsf{if}\ \mathsf{true} < 0\ \mathsf{then\ false\ else\ true}\\
\longrightarrow^*\ &\mathsf{Err}\ 1
\end{aligned}$$

When we run the program, we get a runtime error in the (second) execution of pos when we try to check whether true is a negative number (the error comes with code 1 to indicate that an arithmetic operation failed). Consider what would be required to debug this error. First, we would need to realize that the error occurred 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. Exactly because we haven't inserted the necessary checks, the actual type error occurs in a place not directly related to the real source of the error. Assume, now, that we are the authors of double and that the previous example is the result of a third-party (another programmer) that tries to use our function with their implementation of pos. Let's see how the example would look like, if we had programmed defensively and inserted all the necessary checks in the body of double to help the third-party programmer detect the type error as accurately and as early as possible.

$$\begin{aligned}
&\mathsf{let}\ \mathit{double} = \lambda f.\, \mathsf{if\ is\_fun?}\ f\ \mathsf{then}\\
&\quad \mathsf{let}\ x = f\ 0\ \mathsf{in}\\
&\quad \mathsf{let}\ y = f\ (\mathsf{if\ is\_int?}\ x\ \mathsf{then}\ x\ \mathsf{else\ raise}\ 1)\ \mathsf{in}\\
&\quad \mathsf{if\ is\_int?}\ y\ \mathsf{then}\ y\ \mathsf{else\ raise}\ 1\\
&\quad \mathsf{else\ raise}\ 0\ \mathsf{in}\\
&\mathsf{let}\ \mathit{pos} = \lambda i.\, \mathsf{if}\ i < 0\ \mathsf{then\ false\ else\ true}\ \mathsf{in}\\
&\mathit{double}\ \mathit{pos}
\end{aligned}$$

Now double has all the necessary checks and the example raises an error when the first call to pos returns. That is, because of the checks we inserted, we can at least be sure that nothing is wrong with our implementation of double. However, our previous one-line implementation of double has been replaced with a much more complicated version. Imagine how much more obscure and cluttered the new version would look like if double was doing something more complicated... If only there was a mechanism that (i) let us state a *specification* for double (i.e., that it is a function that consumes a function f from integers to integers and returns an integer) and (ii) injected automatically the implementation of double with all the necessary checks.

Programming languages researchers have developed a language feature that does exactly that: *contracts*. Contracts originate from the programming language Eiffel where they became the corner-stone of the Design by Contract methodology, a methodology that advocates that programmers should start coding by first writing a contract for their code and then code against it. Nowadays, many languages come with built-in or library support for contracts (e.g. Java, Javascript, Python, Ruby, PHP, Racket, Common Lisp). These contracts can describe the specifications not only of functions but also of classes, objects and other language features. In the remainder of this section, we introduce contracts for higher-order functions through a simple calculus.[1]

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 appropriate.

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

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

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

$$\frac{}{(\mathsf{monitor}(v, v_1 \longmapsto v_2))\ v' \longrightarrow \mathsf{raise}\ 3}\ v \neq \lambda x.\, e$$

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

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

[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.

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

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.

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

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 with minimum changes to the code of the initial example, we got an error as informative as in the version where we manually injected the checks. And we achieved this without having to tedioulsy add checks all over our example: contracts enforced automatically and correctly our specification for double and pos.

3.1 Blame

In the example above, the function contract for double helped us correctly and with minimum effort detect that $\mathit{pos}\ 0$ does not evaluate to an integer. However, our calculus for contracts does not provide any information about the source of the contract error. In particular, which part of our program should we blame for this error? Where in the code should we start our debugging effort from to fix the problem? 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?

Assigning blame in our simple example may seem easy but is not. In a higher order setting blame assignment becomes quickly subtle. Since a function can pass from one piece of code to another, the code that ends up applying arguments to a function may be unaware of the function's contract. That is the code during whose execution the contract error is raised may not be the culprit of the contract violation. For instance, assume that our running example consists of three different pieces of code: $double$, pos and the "main" code of the program that applies $double$ to pos. The contract violation occurs when we evaluate the body of $double$ but of course, the error is not $double$'s fault. The contract of $double$ states that its argument should be a function from integers to integers and $double$ uses it as such. So is it pos's fault that it is not a function that returns integers? No, pos's contract specifies that pos should return a boolean, as pos does. Then, if both functions involved are innocent, which piece of code should we blame? The answer is that we should blame "main" because it is the piece of code that brought two functions together without respecting their contracts. Put differently, a contract is a binding agreement between the provider of a function and a piece of code that decides to use it. And this agreement binds the user code even if the user code passes the function to a third piece of code. In practical terms, this interpretation of blame helps us avoid spending time trying to find a bug in $double$ and pos that do nothing more that behave according to their specifications. Instead, blame guides us to reconsider the correctness of the faulty "main" code.

We can extend our simple contracts calculus to keep track of blame, and assign blame correctly upon contract violations. 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 that chooses to use the code with the contract. The idea is that if a function is being monitored, then n is responsible for any code n passes the function to and may apply arguments to the function. We can think of these labels p and n as being module names, or separate pieces of code. In our example, as we discuss above, there are three entities involved, and thus 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.

The rules for blame-tracking monitors are as follows:[3]

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

$$\frac{}{(\mathsf{monitor}(v, v_1 \longmapsto v_2, p, n))\ v' \longrightarrow \mathsf{raise}\ \text{"Blame } p\text{"}}\ v \neq \lambda x.\,e$$

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

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 code responsible for providing to v arguments that satisfy v_1. As in our example, n may not be the immediate context that provides v' but rather the code that agreed to v's contract. By contrast, if the result of the function application doesn't satisfy contract v_2, then p will be blamed.

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 responsible 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 context that uses of the function. In our example, for functions $double$ and pos, this context is the "main" code, i.e., the code that applies $double$ to pos (given these labels, the example raises a contract error that blames "main" as we expect).

[3]Of course, we need to extend the syntax of monitors to include blame labels: $\mathsf{monitor}(e, e, x, x)$. Also we add strings to our values so that we can throw informative exceptions upon contract violations.