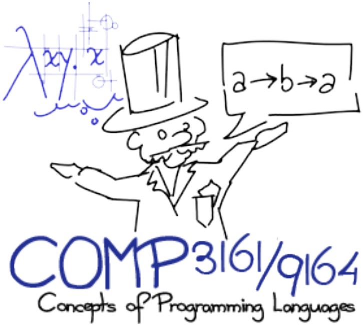

Safety and Liveness; Type Safety; Exceptions

Johannes Åman Pohjola
UNSW
Term 3 2022

Program Properties

Consider a sequence of states, representing the evaluation of a program in a small step semantics (a trace):

$$\sigma_1 \mapsto \sigma_2 \mapsto \sigma_3 \mapsto \cdots \mapsto \sigma_n$$

Some traces are finite, others infinite. To simplify things, we'll make all traces infinite by repeating the final state of any finite behaviour. An infinite sequences of states is called a behaviour.

A property of a program is a set of behaviours.

Safety vs Liveness

1. A safety property states that something **bad~~bad~~** will never happen. For example:

   I will never run out of money.

   These are properties that may be violated by a finite prefix of a behaviour.

2. A liveness property states that something **good~~good~~** will happen. For example:

   If I start drinking now, eventually I will be smashed.

   These are properties that cannot be violated by any finite prefix of a behaviour.

Combining Properties

Safety properties we've seen before

Partial correctness (Hoare Logic)   Static semantics properties

Liveness properties we've seen before

Termination

Theorem

Every property is the intersection of a safety property and a liveness property.

(Not everything of interest is a property. For example, confluence isn't. Why not?)

Types

What sort of properties do types give us?
Adding types to λ-calculus eliminates terms with no normal forms.

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \quad \frac{x : \tau_1, \Gamma \vdash e : \tau_2}{\lambda x.\ e : \tau_1 \rightarrow \tau_2} \quad \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

Remember $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$? Trying to type this requires an infinite type $\tau_1 = \tau_1 \rightarrow \tau_2$.

Theorems

Each well typed λ-term has a normal form (strong normalisation). Furthermore, the normal form has the same type as the original term (subject reduction).

This means that all typed λ-terms terminate!

With Recursion

MinHS, unlike lambda calculus, has built in recursion. We can define terms like:

$$(\mathsf{recfun}\ f :: (\mathtt{Int} \rightarrow \mathtt{Int})\ x = f\ x)\ 3$$

Which has no normal form or final state, despite being typed. **What now?**

The liveness parts of the typing theorems can't be salvaged, but the safety parts can...

Type Safety

Type safety is the property that states:

Well-typed programs do not go wrong.

By “go wrong”, we mean reaching a stuck state—a non-final state with no outgoing transitions.
What are some examples of stuck states?

There are many other definitions of things called “type safety” on the internet. For our purposes, ignore them.

Progress and Preservation

We want to prove that a well-typed program either goes on forever or reaches a final state. We prove this with two lemmas.

How to prove type safety

1. **Progress**, which states that well-typed states are not stuck states. That is, if an expression $e : \tau$ then either e is a final state or there exists a state e' such that $e \mapsto e'$.
2. **Preservation**, which states that evaluating one step preserves types. That is, if an expression $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.

$$e_1 : \tau \overbrace{\mapsto e_2}^{\text{progress}} \underbrace{: \tau}_{\text{preservation}} \overbrace{\mapsto e_3}^{\text{progress}} \underbrace{: \tau}_{\text{preservation}} \overbrace{\mapsto}^{\text{progress}} \cdots$$

In the real world

Which of the following languages are type safe?

- C
- C++
- Haskell
- Java
- Python
- Rust
- MinHS

Why is MinHS not type safe?

Division by Zero

We can assign a type to a division by zero:

$$\frac{\dfrac{}{(\texttt{Num}\ 3) : \texttt{Int}} \qquad \dfrac{}{(\texttt{Num}\ 0) : \texttt{Int}}}{(\texttt{Div}\ (\texttt{Num}\ 3)\ (\texttt{Num}\ 0)) : \texttt{Int}}$$

But there is no outgoing transition from this state (nor is it final)!
$\Rightarrow$ We have violated progress.
We have two options:

1. **Change the static semantics** to exclude division by zero. This reduces to the halting problem, so we would be forced to overapproximate.
2. **Change the dynamic semantics** so that the above state has an outgoing transition.

Our Cop-Out

Add a new state, `Error`, that is the successor state for any partial function:

$$\frac{}{(\texttt{Div}\ v\ (\texttt{Num}\ 0)) \mapsto_M \texttt{Error}}$$

Any state containing `Error` evaluates to `Error`:

$$\frac{}{(\texttt{Plus}\ e\ \texttt{Error}) \mapsto_M \texttt{Error}} \qquad \frac{}{(\texttt{Plus}\ \texttt{Error}\ e) \mapsto_M \texttt{Error}}$$

$$\frac{}{(\texttt{If}\ \texttt{Error}\ t\ e) \mapsto_M \texttt{Error}}$$

(and so on – this is much easier in the C machine!)

Type Safety for Error

We've satisfied progress by making a successor state for partial functions, but how should we satisfy preservation?

$$\frac{}{\texttt{Error} : \tau}$$

That's right, we give `Error` <u>any</u> type.

Dynamic Types

Some languages (e.g. Python, JavaScript) are called dynamically typed. We call these unityped, as they achieve type safety with a trivial type system containing only one type, here written $\star$:[1]

$$\frac{}{\Gamma \vdash e : \star}$$

They achieve type safety by defining execution for every syntactically valid expression, even those that are not well typed.

[1] The things these languages call types are part of values. They aren't types.

Exceptions

`Error` may satisfy type safety, but it's not satisfying as a programming language feature. When an error occurs, we may want a way to recover. We will add more fine grained error control – exceptions – to MinHS.

Example (Exceptions)

`try`/`catch`/`throw` in Java, `setjmp`/`longjmp` in C, `try`/`except`/`raise` in Python.

Exceptions Syntax

	Raising an Exception	Handling an Exception
Concrete	raise e	try e_1 handle $x \Rightarrow e_2$
Abstract	$(\texttt{Raise}\ e)$	$(\texttt{Try}\ e_1\ (x.\ e_2))$

Informal Semantics

Example

```
try
   if $y \leq 0$ then
      raise DivisorError
   else
      $(x/y)$
handle $err \Rightarrow -1$
```

For an expression ($\textbf{try}\ e_1\ \textbf{handle}\ x \Rightarrow e_2$) we

1. Evaluate e_1
2. If **raise** v is encountered while evaluating e_1, we bind v to x and evaluate e_2.

Note that it is possible for **try** expressions to be nested.

- The inner-most **handle** will catch exceptions.
- Handlers may re-raise exceptions.

Static Semantics

The type given to exception values is usually some specific blessed type τ_{E} that is specifically intended for that purpose. For example, the `Throwable` type in Java. In dynamically typed languages, the type is just the same as everything else (i.e. $\star$).

Typing Rules

$$\frac{\Gamma \vdash e : \tau_{\mathsf{E}}}{\Gamma \vdash (\texttt{Raise}\ e) : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \quad x : \tau_{\mathsf{E}}, \Gamma \vdash e_2 : \tau}{\Gamma \vdash (\texttt{Try}\ e_1 (x.\ e_2)) : \tau}$$

Dynamic Semantics

Easier to describe using the C Machine. We introduce a new type of state, $s \prec\!\!\!\prec v$, that means an exception value v has been raised. The exception is bubbled up the stack s until a handler is found.

Evaluating a `Try` Expression

$$s \succ (\texttt{Try}\ e_1\ (x.\ e_2)) \mapsto_C (\texttt{Try}\ \Box\ (x.\ e_2)) \triangleright s \succ e_1$$

Returning from a `Try` without raising

$$(\texttt{Try}\ \Box\ (x.\ e_2)) \triangleright s \prec v \mapsto_C s \prec v$$

Evaluating a `Raise` expression

$$s \succ (\texttt{Raise}\ e) \mapsto_C (\texttt{Raise}\ \Box) \triangleright s \succ e$$

Raising an exception

$$(\texttt{Raise}\ \Box) \triangleright s \prec v \mapsto_C s \prec\!\!\!\prec v$$

Catching an exception

$$(\texttt{Try}\ \Box\ (x.\ e_2)) \triangleright s \prec\!\!\!\prec v \mapsto_C s \succ e_2[x := v]$$

Propagating an exception

$$f \triangleright s \prec\!\!\!\prec v \mapsto_C s \prec\!\!\!\prec v$$

Efficiency Problems

The approach described above is highly inefficient. Throwing an exception takes linear time with respect to the depth of stack frames!
Only the most simplistic implementations work this way. A more efficient approach is to keep a separate stack of handler frames.

Handler frames

A handler frame contains:

1. A copy of the control stack above the `Try` expression.
2. The exception handler that is given in the `Try` expression.

We write a handler frame that contains a control stack s and a handler $(x.\ e_2)$ as $(\texttt{Handle}\ s\ (x.\ e_2))$.

Efficient Exceptions

Evaluating a `Try` now pushes the handler onto the handler stack and a marker onto the control stack.

$$(h, s) \succ (\texttt{Try}\ e_1\ (x.\ e_2)) \mapsto_C (\texttt{Handle}\ s\ (x.\ e_2)\ \triangleright h, (\texttt{Try}\ \square) \triangleright s) \succ e_1$$

Returning without raising in a `Try` block removes the handler again:

$$(\texttt{Handle}\ s\ (x.\ e_2)\ \triangleright h, (\texttt{Try}\ \square) \triangleright s) \prec v \mapsto_C (h, s) \prec v$$

Raising an exception now uses the handler stack to immediately jump to the handler:

$$(\texttt{Handle}\ s\ (x.\ e_2)\ \triangleright h, (\texttt{Raise}\ \square) \triangleright s') \prec v \mapsto_C (h, s) \succ e_2[x := v]$$

Exceptions in Practice

Exceptions are useful, but they are a form of non-local control flow and should be used carefully.
In Haskell, exceptions tend to be avoided as they make a liar out of the type system:

$$\mathit{head} :: [a] \to a$$

In Java, checked exceptions allow the possibility of exceptions to be tracked in the type system.

Monads

One of the most common uses of the Haskell monad construct is for a kind of error handling that is honest about what can happen in the types.