COMP3161/COMP9164 Supplementary Lecture Notes

# Type Safety and Exceptions

Liam O'Connor

November 1, 2019

When we define a static semantics for a language, we wish that static semantics to imply some properties about the dynamic semantics. In this notes, we will discuss what *properties* are, how we can classify them, and the kinds of properties we can ensure from static semantics like type systems. Lastly, we will extend MinHS with *exceptions*, an error-handling mechanism, which allows us to ensure the property of *type safety* in the presence of partial functions.



## 1 Properties

A *trace* or *execution* of a program is a (possibly infinite) sequence of states $\sigma_1 \mapsto \sigma_2 \mapsto \cdots \mapsto \sigma_n$ that follows the execution of a program in small step semantics.

A *behaviour* is like a trace, however we generalise the notion slightly so as to only have to deal with infinite sequences of states. A finite sequence of states can be made into a behaviour simply by repeating the final state forever when the program terminates.

With the definition of behaviours, we can define a *property* formally as a *set of behaviours*. A very simple property would be termination, expressed formally as $\{b \mid \exists i.\ b_i \in F\}$, where $b_i$ refers to the $i$th state in the behaviour $b$, and $F$ refers to the set of final states.

### 1.1 Safety and Liveness

There are generally two ways to classify properties:

1. A *safety* property states that something **bad** does not happen. For example:

   *I will never run out of money.*

   Formally, these are those properties that may be violated by a *finite prefix* of a behaviour. For example, if I spend all my money at the pub and run out of money, then I have taken a finite sequence of steps that violates the property. Examples of safety properties we've seen before include hoare triples $\{\varphi\}s\{\psi\}$, and many of the static semantics properties we've checked (e.g. that variables are initialised before they're used, and that all variables used are in scope).

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

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

   These are properties that cannot be violated by a finite prefix of a behaviour — there is always some way to satisfy the property after any finite number of steps. For example, even if I drink 100 beers and am still not intoxicated, I could always get drunk on the 101st beer. So there is no telling that the property has been violated no matter how many steps I've already taken, as I could always satisfy the property later. Examples of liveness properties we've seen before include termination, and also the confluence of $\beta$-reduction.

A very powerful result from Alpern and Schneider[1] is that *all properties* are the intersection of some safety and some liveness property. For example, the property that "the program returns the number three" is the intersection of the liveness property that the program returns a value (as opposed to looping forever), and the safety property that says that any returned value of the program should be three.

## 1.2 Type Safety

A *type system* is a type of static semantics used for verifying programs and improving the reliability of software. It is, essentially, a means of annotating expressions and values in a program with a tag, called a *type*, which tells us something about the set of runtime data the expression can represent.

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



Adding types to $\lambda$-calculus, as shown above, ensures two major properties. Firstly, and most significantly, *all terms will reduce to a normal form.* Terms such as $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$, which has no normal form, cannot be assigned a type under these rules (try it and see for yourself ☺). Furthermore, we also know that the normal form of each term will have the same type as the original term.



If we look at a language like MinHS, however, we have built-in recursion in the form of the **recfun** construct. A term like



$$(\textbf{recfun}\ f :: (\texttt{Int} \to \texttt{Int})\ x = f\ x)\ 3$$

will clearly loop forever. So, we don't get the guarantees from MinHS's type system that we get from adding types to $\lambda$-calculus, as the above term has a type, but no normal form.



It turns out that while we can't guarantee the *liveness* part of the typing properties, we can guarantee the *safety* part. This is a property called *type safety.*

Succinctly, it can be stated as:

*Well typed programs do not go wrong.*



By "go wrong", we mean reaching a *stuck state* — that is, a non-final state with no outgoing transitions.

We can decompose type safety into two sub-lemmas:

A language with small-step states $\Sigma$, final states $F \subseteq \Sigma$, state transition relation $\mapsto$, and typing rules is *type safe* if it has two properties:

- *Progress* - If a program can be typed, it is either a final state or can progress to another state. That is, if $\vdash e : \tau$ then $e \in F$ or $\exists e'.\ e \mapsto e'$.
- *Preservation* - If a program has a type, evaluation will not change that type. That is, if $\vdash e : \tau$ and $e \mapsto e'$ then $\vdash e' : \tau$.

It can be seen from the above definition that well typed programs will not reach a stuck state. If the program is a final state, then it is by definition not stuck. If not, we know from the progress property that the program must move to a new state. We know from preservation that this new state is also typed, which means (from progress) that it must either be a final state or progress to a new state. Similar reasoning applies until the program terminates (or loops).

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

[1] It's a readable paper if you're familiar with metric spaces. `https://www.cs.cornell.edu/fbs/publications/defliveness.pdf`

It therefore follows that languages such as C, which are *unsafe*, could reach a stuck state. In such a situation, the program doesn't simply *halt* (or at least, it's not obliged to). What happens is left *undefined*. For example, there is no telling what this C program will do without referring to platform or compiler documentation:

```
int main() {
  return *((int*)(0x0));
}
```

Clearly, speaking of type safety is only applicable in the context of formal treatment of programming languages. Determining exactly what guarantees a type system gives you requires these techniques.

In general, the more expressive the type system is, the more information can be inferred by the compiler. Therefore, for practical reasons we typically want type checking to be decidable. If it was not, our compiler may not terminate. There are languages such as C++, however, where type checking may not terminate.



## 2 Dealing with Partiality



Suppose we have a partial operation, such as division, typed as follows:

$$\frac{\Gamma \vdash t_1 : \texttt{Int} \quad \Gamma \vdash t_2 : \texttt{Int}}{\Gamma \vdash \texttt{Div}\ t_1\ t_2 : \texttt{Int}}$$



We've assigned it a type, but it does not necessarily guarantee progress. The expression $\texttt{Div}\ x\ 0$ for any $x$ will not return a value. There are two solutions:



- *Change the static semantics* - That is, disallow divisions by zero statically. There are techniques to approximate this for turing-complete languages, however in general this is undecidable. For those that are interested, the proof is a corollary of Rice's theorem.
- *Change the dynamic semantics* - This approach is used by most languages.



Seeing as MinHS is turing complete, we are unable to statically analyse if the program divides by zero. Hence, we shall extend the dynamic semantics of the language to handle the situation at runtime.

The simplest fix is to make partial functions yield some new state $\texttt{error} \in F$ for undefined cases:

$$\frac{}{\texttt{Div}\ v\ (\texttt{Num}\ 0) \mapsto \texttt{error}}$$

Furthermore, we would define `error` to interrupt any nested computation and produce `error`.

$$\frac{}{\texttt{Plus}\ \texttt{error}\ e \mapsto \texttt{error}}$$

$$\frac{}{\texttt{Plus}\ e\ \texttt{error} \mapsto \texttt{error}}$$

$$\frac{}{\texttt{If}\ \texttt{error}\ e_1\ e_2 \mapsto \texttt{error}}$$

There are, of course, a very large number of additional `error` propagation rules. Here, our abstract machines actually buy us some brevity. We simply state that partial functions result in `error`, and completely annihilate the stack (e.g in the *C Machine*):

$$\frac{}{\texttt{Div}\ v\ \square \triangleright s \prec 0\ \mapsto\ \texttt{error}}$$

This guarantees *progress* - partial functions will evaluate to `error` where they are not defined, meaning that the evaluation will not hit a stuck state.

We have yet to ensure *preservation*, however. Preservation says that type is preserved across evaluation. Seeing as any partial function application (of any type) could evaluate to `error`, the only way to make `error` respect preservation is to make it a member of *every* type:

$$\frac{}{\Gamma \vdash \texttt{error} : \tau}$$

## 2.1 Exceptions

Adding a `error` state seems well and good for ensuring type safety, but many real-world languages have more robust, fine-grained error handling techniques, namely exceptions.

Exceptions are a means for a function to exit without returning. Instead, the function may *raise* an exception, which is caught by an exception handler somewhere further up the runtime stack. Most of you would have seen exceptions from languages such as Java, Python, or C++.

We will extend MinHS to include exceptions by adding two pieces of abstract syntax: $\texttt{Try}\ e_1\ x.e_2$ will evaluate $e_1$, and if $\texttt{Raise}\ \tau\ v$ is ever encountered while evaluating $e_1$, it will stop evaluating $e_1$, and start evaluating $e_2$ where $x$ is bound to the value of $v$.

These `Try` expressions can of course be nested, and exceptions can be re-`Raise`d within an exception handler.

Exception values (such as $v$ in the above example), are made to be of a fixed type, $\tau_{exc}$. It is not relevant what type this is, it could be a special `Exception` type, it could be an `Int` error code, or just a `String` message.

We type these new expressions as follows. `Try` expressions take the type of their subexpressions, and `raise` expressions are of any type specified in the expression (for a similar reason to the typing of `error`):

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

### 2.1.1 Dynamic Semantics for the *C Machine*

We introduce a new execution mode (in addition to $\prec$ and $\succ$), used to signify that an exception is being raised, written $\preccurlyeq$.

So, when we evaluate a `try` expression, we simply evaluate the first subexpression, and push the handler onto the stack:

$$\frac{}{s \succ \texttt{Try}\ e_1\ x.e_2 \mapsto_c \texttt{Try}\ \square\ \ x.e_2 \rhd s \succ e_1}$$

Then, if the evaluation returns, we simply discard the `try` stack frame:

$$\frac{}{\texttt{Try}\ \square\ \ x.e_2 \rhd s \prec v \mapsto_c s \prec v}$$

If we encounter a `raise` expression, we first evaluate the exception value being raised:

$$\frac{}{s \succ \texttt{Raise}\ \tau\ e \mapsto_c \texttt{Raise}\ \tau\ \square \rhd s \succ e}$$

And, once it returns, we enter the new exception handling mode, $\preccurlyeq$:

$$\frac{}{\texttt{Raise}\ \tau\ \square \rhd s \prec v \mapsto_c s \preccurlyeq v}$$

This mode continuously pops frames off the stack:

$$\frac{}{f \rhd s \preccurlyeq v \mapsto_c s \preccurlyeq v}$$

Until at last we encounter a `Try` expression, where the handler is evaluated.

$$\frac{}{\texttt{Try}\ \square\ \ x.e_2 \rhd s \preccurlyeq v \mapsto_c s \succ e_2[x := v]}$$

### 2.1.2 Optimising Exceptions

The problem with this approach is one of performance. Raising an exception is $O(n)$ in the size of the stack, which could be a serious performance hit if the stack is very large (for example, in a big recursive function).

Seeing as in our abstract machines we are concerned about performance, we will refine our machine definition to make exception handling fast.

We will define a new type of stack, a *Handler* stack. The empty handler stack is denoted by $\star$, and each handler frame consists of a runtime stack, and the handler expression:

$$\frac{}{\star\ \mathit{HStack}} \qquad \frac{s\ \mathit{HStack} \quad e\ \mathit{Expr} \quad r\ \mathit{Stack}}{\langle r, x.e\rangle \triangleright s\ \mathit{HStack}}$$

Our states will now resemble $h, r \succ e$, where $h$ is the handler stack, $r$ is the runtime stack. The "exception handling" mode $\prec$ is not needed and therefore is removed.

When we enter a `Try` block, we add a handler to the handler stack, and a placeholder to the runtime stack. The handler includes the current runtime stack and the handler expression:

$$\frac{}{h, r \succ \mathtt{Try}\ e_1\ x.e_2 \ \mapsto_c\ \langle r, x.e_2\rangle \triangleright h, \mathtt{Try} \triangleright r \succ e_1}$$

We include the placeholder so that if we return from a `Try` block, we can remove the handler from the handler stack as it was not used:

$$\frac{}{\langle r', x.e_2\rangle \triangleright h, \mathtt{Try} \triangleright r \prec v \ \mapsto_c\ h, r \prec v}$$

When we encounter a `Raise` expression, we first evaluate the exception value as normal, and then we immediately switch to the runtime stack in the top handler frame of the exception stack, saving us the trouble of manually going through the runtime stack frame by frame:

$$\frac{}{\langle r', x.e_2\rangle \triangleright h, \mathtt{Raise}\ \square \triangleright r \prec v \ \mapsto_c\ h, r' \succ e_2[x := v]}$$

*Note*: It may seem inefficient to copy the runtime stack to the handler stack each time a `Try` block is reached. Note however that, in the course of evaluating the try block, the machine will never pop off the `Try` placeholder. Therefore a pointer to the current runtime stack could be kept in the handler stack rather than a copy. Everything above that pointer is freed when an exception is raised.