Progress and Preservation of Typed Programs

Viktor Kunčak

Getting stuck according to semantics

If a term t makes no sense, our operational semantics will have no rule to define its evaluation, so there is no t' such that $t \rightsquigarrow t'$

Example: consider this expression:

$$\textbf{if } (5)\ 3\ \textbf{else}\ 7$$

the expression 5 cannot be evaluated further and is a constant, but there are no rules for when condition of **if** is a number constant; there are only such rules for boolean constants.

Such terms, that are not constants and have no applicable rules, are called **stuck**, because no further steps are possible.

Stuck terms indicate errors. Type checking is a way to detect them **statically**, without trying to (dynamically) execute a program and see if it will get stuck or produce result.

Type Judgement

We want to know if errors happen in the sequence

$$t_1 \rightsquigarrow t_2 \rightsquigarrow t_3 \rightsquigarrow \ldots$$

but we do not want to run the program to find all the $t_2, t_3, \ldots$

Instead, we **approximate** program execution by computing **types** that $t_1, t_2, t_3, \ldots$ may have and use this information to conclude that no errors can happen.

We write that an expression (term) t **type checks and has type** τ using notation

$$t : \tau$$

Like relation $\leq$, the colon symbol **:** is a binary relation.
We define it **inductively**, using **inference rules**.

Type checking rule for **if** expression

$$\frac{b : \mathit{Bool}, \quad t_1 : \tau, \quad t_2 : \tau}{(\textbf{if } (b)\ t_1 \textbf{ else } t_2) : \tau}$$

We read it like this: WHEN

- the expression b type checks and has type Bool, and
- the expression t_1 type checks and has some type, τ, and
- the expression t_2 type checks and has **the same** type τ

THEN

- the expression $(\textbf{if } (b)\ t_1 \textbf{ else } t_2)$ also type checks and has type τ

This is the only rule for **if**, so we cannot conlude that $(\textbf{if } (5)\ 3 \textbf{ else } 7) : \tau$ for some τ.
We say that $(\textbf{if } (5)\ 3 \textbf{ else } 7)$ does not type check.

Type Rule for Constants and Operations

All special case of function application: given arguments must match the declared parameters:

$$\frac{f \;:\; (\tau_1 \times \cdots \times \tau_n) \rightarrow \tau_0, \qquad t_1 : \tau_1, \quad \ldots \;, \quad t_n : \tau_n}{f(t_1, \ldots, t_n) : \tau_0}$$

We treat primitives like applications of functions e.g.

$$\begin{array}{ccl} + & : & Int \times Int \rightarrow Int \\ \leq & : & Int \times Int \rightarrow Bool \\ \&\& & : & Bool \times Bool \rightarrow Bool \end{array}$$

so a special case is, e.g.,

$$\frac{+ \;:\; (Int \times Int) \rightarrow Int, \qquad t_1 : Int, \quad t_2 : Int}{(t_1 + t_n) : Int}$$

From Binary to Ternary Relation: Type Environment

If x is a parameter, we cannot determine whetehr $x : \mathit{Int}$ or $x : \mathit{Bool}$ without knowing the declared type of x.
To specify the types of identifiers, we use a partial function that maps identifiers to their types. We usually denote it with Γ.

Instead of a binary relation $t : \tau$, we therefore use a **ternary relation**:

$$\Gamma \vdash t : \tau$$

meaning:

In the type environment Γ, term t type checks and has type τ.

The typing relation relates three things: Γ, t, τ.
We could have written $(\Gamma, t, \tau) \in R$ for some relation R, but we choose to write $\Gamma \vdash t : \tau$ (this is just a matter of notation).

Type Checking Rules with Environment

Instead of

$$\frac{b : Bool, \quad t_1 : \tau, \quad t_2 : \tau}{(\textbf{if } (b) \ t_1 \textbf{ else } t_2) : \tau}$$

the rule for **if** becomes:

$$\frac{\Gamma \vdash b : Bool, \quad \Gamma \vdash t_1 : \tau, \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash (\textbf{if } (b) \ t_1 \textbf{ else } t_2) : \tau}$$

The rule for function application becomes:

$$\frac{\Gamma \vdash f : \tau_1 \times \dots \times \tau_n \to \tau_0, \quad \Gamma \vdash t_1 : \tau_1, \ \dots, \ \Gamma \vdash t_n : \tau_n}{\Gamma \vdash f(t_1, \dots, t_n) : \tau_0}$$

Now we can give rule for parameters:

$$\frac{(x, \tau) \in \Gamma}{\Gamma \vdash x : \tau}$$

Constants are easy anyway:

$$\frac{}{\Gamma \vdash 42 : Int} \qquad \frac{}{\Gamma \vdash true : Bool}$$

Type Checking the Factorial Body

Let $\Gamma = \{(n, Int), (fact, Int \rightarrow Int)\}$

$$
\dfrac{
  \dfrac{
    \dfrac{(n, Int) \in \Gamma}{\Gamma \vdash n : Int \quad \Gamma \vdash 1 : Int}
  }{\Gamma \vdash n \leq 1 : Bool,}
  \qquad \Gamma \vdash 1 : Int \qquad
  \dfrac{
    \dfrac{(fact, Int \rightarrow Int) \in \Gamma}{\Gamma \vdash fact : Int \rightarrow Int}
    \quad
    \dfrac{\dfrac{(n : Int) \in \Gamma}{\Gamma \vdash n : Int} \quad \Gamma \vdash 1 : Int}{\Gamma \vdash n - 1 : Int}
  }{\Gamma \vdash fact(n-1) : Int}
}{\Gamma \vdash (\textbf{if } (n \leq 1) \ 1 \textbf{ else } n * fact(n-1)) : Int}
$$

We applied given type rules and created a derivation tree to show that the final expression type checks and has type Int.

Observation on Replacing Sub-Trees

Let $\Gamma = \{(n, Int), (fact, Int \rightarrow Int)\}$

$$
\begin{array}{c}
\hspace{14em} (n : Int) \in \Gamma \\
\hline
(n, Int) \in \Gamma \qquad\qquad (fact, Int \rightarrow Int) \in \Gamma \quad \Gamma \vdash n : Int \quad \Gamma \vdash 1 : Int \\
\hline
\Gamma \vdash n : Int \quad \Gamma \vdash 1 : Int \qquad \Gamma \vdash fact : Int \rightarrow Int \qquad \Gamma \vdash n-1 : Int \\
\hline
\Gamma \vdash n \leq 1 : Bool, \quad \Gamma \vdash 1 : Int \quad \Gamma \vdash fact(n-1) : Int \\
\hline
\Gamma \vdash (\textbf{if } (n \leq 1) \ 1 \ \textbf{else } n * fact(n-1)) : Int
\end{array}
$$

Suppose we replace $n : Int$ with $4 : Int$.
Types of n and 4 are the same (Int), so we obtain a valid tree:

$$
\begin{array}{c}
\hspace{8em} (fact, Int \rightarrow Int) \in \Gamma \quad \Gamma \vdash 4 : Int \quad \Gamma \vdash 1 : Int \\
\hline
\Gamma \vdash 4 : Int \quad \Gamma \vdash 1 : Int \qquad \Gamma \vdash fact : Int \rightarrow Int \qquad \Gamma \vdash 4-1 : Int \\
\hline
\Gamma \vdash 4 \leq 1 : Bool, \quad \Gamma \vdash 1 : Int \quad \Gamma \vdash fact(4-1) : Int \\
\hline
\Gamma \vdash (\textbf{if } (4 \leq 1) \ 1 \ \textbf{else } 4 * fact(4-1)) : Int
\end{array}
$$

How to Type Check a Program

Given initial program (e,t) (e are definitions and t is main level expression), define

$$\Gamma_0 = \{(f, \tau_1 \times \cdots \times \tau_n \to \tau_0) \mid (f, \_, (\tau_1, \ldots, \tau_n), t_f, \tau_0) \in e\}$$

We say program type checks iff:
(1) the top-level expression type checks:

$$\Gamma_0 \vdash t : \tau$$

and
(2) each function body type checks:

$$\Gamma_0 \cup \{(x_1, \tau_1), \ldots, (x_n, \tau_n)\} \vdash t_f : \tau_0$$

for each $(f, (x_1, \ldots, x_n), (\tau_1, \ldots, \tau_n), t_f, \tau_0) \in e$

Type Checking Factorial Program

$p_{fact} = (e, fact(2))$
where $e(fact) = (n, Int,\ \textbf{if}\ (n \leq 1)\ 1\ \textbf{else}\ n * fact(n-1), Int)$

$$\Gamma_0 = \{(n, Int \to Int)\}$$

The program type checks iff:
(1) the top-level expression type checks:

$$\Gamma_0 \vdash fact(2) : \tau$$

and
(2) the body of the function (here there is only one) type checks to the declared result of the function:

$$\Gamma_0 \cup \{(n, Int)\} \vdash \textbf{if}\ (n \leq 1)\ 1\ \textbf{else}\ n * fact(n-1) : Int$$

When type checking the body, we add the types of parameters into the environment.

Soundness through progress and preservation

Soundness theorem: *if program type checks, its evaluation does not get stuck*.
Proof uses the following two lemmas (a common approach):

- progress: if a program type checks, it is not stuck: if

  $$\Gamma \vdash t : \tau$$

  then either t is a constant (execution is done) or there exists t' such that $t \rightsquigarrow t'$
- preservation: if a program type checks and makes one $\rightsquigarrow$ step,
  then the result again type checks
  in our simple system, it type checks *and has the same type*: if

  $$\Gamma \vdash t : \tau$$

  and $t \rightsquigarrow t'$ then

  $$\Gamma \vdash t' : \tau$$

Proof of progress and preservation - case of if

We prove conjunction of progress and preservation by induction on term t such that $\Gamma \vdash t : \tau$. The operational semantics defines the non-error cases of an interpreter, which enables case analysis. Consider the case when t is $\textbf{if}\ (b)\ t_1\ \textbf{else}\ t_2$. By type checking rules, this can only type check if the condition b type checks and has type Bool. By inductive hypothesis and progress *either b is a constant or it can be reduced to a b'*. If it is constant one of these rules apply (so we get progress):

$$\frac{}{(\textbf{if}\ (\mathit{true})\ t_1\ \textbf{else}\ t_2) \rightsquigarrow t_1}$$

$$\frac{}{(\textbf{if}\ (\mathit{false})\ t_1\ \textbf{else}\ t_2) \rightsquigarrow t_2}$$

and the result, by type rule for **if**, has type τ (preservation). If b is not constant, then it reduces to b', so the assumption of the rule

$$\frac{b \rightsquigarrow b'}{(\textbf{if}\ (b)\ t_1\ \textbf{else}\ t_2) \rightsquigarrow (\textbf{if}\ (b')\ t_1\ \textbf{else}\ t_2)}$$

applies, and hence t also makes progress; denote the result t'. By preservation IH, b' also has type Bool, so we can derive $t' : \tau$, re-using the type derivations for t_1 and t_2.

Progress and preservation - user defined functions

Following the cases of operational semantics, either all arguments of a function have been evaluated to a constant, or some are not yet constant.
If they are not all constants, the case is as for the condition of **if**, and we establish progress and preservation analogously.
Otherwise rule

$$\frac{}{f(c_1, \ldots, c_n) \rightsquigarrow t_f[x_1 := c_1, \ldots, x_n := c_n]}$$

applies, so progress is ensured. For preservation, we need to show

$$\Gamma \vdash t_f[x_1 := c_1, \ldots, x_n := c_n] : \tau \qquad (*)$$

where $e(f) = ((x_1, \ldots, x_n), (\tau_1, \ldots, \tau_n), t_f, \tau_0)$ and t_f is the body of f. According to type rules $\tau = \tau_0$ and $\Gamma \vdash c_i : \tau_i$.

Progress and preservation - substitution and types

Function f definition type checks, so $\Gamma' \vdash t_f : \tau_0$ where $\Gamma' = \Gamma \oplus \{(x_1, \tau_1), \ldots, (x_n, \tau_n)\}$. Consider the type derivation tree for t_f and replace each use of $\Gamma' \vdash x_i : \tau_i$ with $\Gamma \vdash c_i : \tau_i$. By our Observation on Replacing Subtrees, the result is a type derivation for $(*)$:

$$\Gamma \vdash t_f[x_1 := c_1, \ldots, x_n := c_n] : \tau \qquad (*)$$

Therefore, the preservation holds in this case as well.