IIT CS534: Types and Programming Languages

E Static Semantics and Type Safety

Stefan Muller

Tuesday, Sept. 2

1 Static Semantics (Type System)

Recalling the E language from last time, there are syntactically valid expressions that don't step. For example, $\overline{1} + \text{“}Hello\text{”}$. We say such expressions are "stuck." There are also valid expressions that will step, but will (eventually) step to expressions that can no longer step (these will "get stuck"). For example, $(\overline{1} + \overline{2}) + \text{“}Hello\text{”} \mapsto \overline{3} + \text{“}Hello\text{”}$.

We fix this with a type system! We also call this the "static semantics" of the language because it's also a way of assigning meaning to a program (even just as simple as whether the program will evaluate properly) statically.

We have a syntax for types:

$$\begin{array}{rcll} \tau & ::= & \mathsf{int} & \text{Integer type} \\ & | & \mathsf{string} & \text{String type} \end{array}$$

The typing judgment $e : \tau$ means "e has type τ."

$$\frac{}{\overline{n} : \mathsf{int}}\ (\textsc{TypeNum}) \qquad \frac{}{\text{“}s\text{”} : \mathsf{string}}\ (\textsc{TypeString}) \qquad \frac{e_1 : \mathsf{int} \qquad e_2 : \mathsf{int}}{e_1 + e_2 : \mathsf{int}}\ (\textsc{TypeAdd})$$

$$\frac{e_1 : \mathsf{string} \qquad e_2 : \mathsf{string}}{e_1 \hat{} e_2 : \mathsf{string}}\ (\textsc{TypeCat}) \qquad \frac{e : \mathsf{string}}{|e| : \mathsf{int}}\ (\textsc{TypeLen})$$

We can build derivations to show that expressions are well-typed (abbreviating TypeNum as TN so it fits)

$$\dfrac{\dfrac{}{\overline{0} : \mathsf{int}}\ (\text{TN}) \qquad \dfrac{\dfrac{\dfrac{}{\overline{1} : \mathsf{int}}\ (\text{TN}) \qquad \dfrac{}{\overline{2} : \mathsf{int}}\ (\text{TN})}{\overline{1} + \overline{2} : \mathsf{int}}\ (\textsc{TypeAdd}) \qquad \dfrac{}{\overline{3} : \mathsf{int}}\ (\text{TN})}{(\overline{1} + \overline{2}) + \overline{3} : \mathsf{int}}\ (\textsc{TypeAdd})}{\overline{0} + (\overline{1} + \overline{2}) + \overline{3} : \mathsf{int}}\ (\textsc{TypeAdd})$$

This doesn't prevent us from writing stuck expressions: $\overline{1} + \text{“}Hello\text{”}$ is still syntactically valid (and will continue to be), but it doesn't have a type, that is, there is no τ such that $\overline{1} + \text{“}Hello\text{”} : \tau$. The idea is that if e is well-typed, that is, if $e : \tau$ for some τ, then e will never "get stuck". We now formalize this as an idea called *type safety*.

2 Type Safety

The slogan of type safety, attributed to Robin Milner, is "well-typed programs can't go wrong." Type safety is the property that guarantees this: you can see it as a proof that your type system achieves the goal it sets

out to. In the context described above (and in most practical type systems), "go wrong" is understood to mean "get stuck" or encounter a run-time type error. We typically prove type safety as two theorems:

- **Progress** says that if e is well-typed, it's a value (i.e., done evaluating) or can take a step. In other words, it's not stuck.
- **Preservation** says that if a well-typed expression takes a step, the new expression is well-typed (with the same type).

Together, these show that a well-typed program won't *become* stuck. For example, if $e_1 : \tau$ and, by progress, $e_1 \mapsto e_2$, then, by preservation, $e_2 : \tau$, so by progress, $e_2 \mapsto e_3$, and so on.

Theorem 1 (Preservation)**.** *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

Proof. By induction on the derivation of $e \mapsto e'$.

- StepAdd. Then $e = \overline{n_1} + \overline{n_2}$ and $e' = \overline{n_1 + n_2}$. By inversion on TypeAdd, we have $\tau = \mathsf{int}$.[1] By TypeNum, $e' : \mathsf{int}$.
- StepCat. Then $e =$ "s_1" ^ "s_2" and $e' =$ "$s_1 s_2$". By inversion on TypeCat, $\tau = \mathsf{string}$. By TypeString, $e' : \mathsf{string}$.
- StepLen. Then $e = |\text{"}s\text{"}|$ and $e' = \overline{|s|}$. By inversion on TypeLen, $\tau = \mathsf{int}$. By TypeNum, $e' : \mathsf{int}$.
- StepSearchAddLeft. Then $e = e_1 + e_2$ and $e' = e_1' + e_2$ and $e_1 \mapsto e_1'$. By inversion on TypeAdd, $\tau = \mathsf{int}$ and $e_1 : \mathsf{int}$ and $e_2 : \mathsf{int}$. By induction, $e_1' : \mathsf{int}$. By TypeAdd, $e' : \mathsf{int}$.
- StepSearchAddRight. Then $e = \overline{n_1} + e_2$ and $e' = \overline{n_1} + e_2'$ and $e_2 \mapsto e_2'$. By inversion on TypeAdd, $\tau = \mathsf{int}$ and $\overline{n_1} : \mathsf{int}$ and $e_2 : \mathsf{int}$. By induction, $e_2' : \mathsf{int}$. By TypeAdd, $e' : \mathsf{int}$.
- StepSearchCatLeft. Basically the same as the case for StepSearchAddLeft.
- StepSearchCatRight. Basically the same as the case for StepSearchAddRight.
- StepSearchLen. Then $e = |e_0|$ and $e' = |e_0'|$ and $e_0 \mapsto e_0'$. By inversion on TypeLen, $\tau = \mathsf{int}$ and $e_0 : \mathsf{string}$. By induction, $e_0' : \mathsf{string}$. By TypeLen, $e' : \mathsf{int}$.

□

We need the following lemma to prove Preservation:

Lemma 1. *Canonical Forms*

1. *If e val and $e : \mathsf{int}$, then $e = \overline{n}$ for some n.*
2. *If e val and $e : \mathsf{string}$, then $e =$ "s" for some s.*

Proof. 1. By "induction"[2] on the derivation of e val.

- ValNum. Then $e = \overline{n}$.
- ValString. Doesn't apply because this would mean $e =$ "s" and that doesn't have type int.

2. Similar.

□

Theorem 2 (Progress)**.** *If $e : \tau$, then e val or there exists e' such that $e \mapsto e'$.*

[1] This is a proof technique we haven't seen before. Inversion means using the inference rules "upside down." We want to know what τ is, so we look at the typing rules and see all the possible ways we could have derived the fact that $e : \tau$, that is, that $\overline{n_1} + \overline{n_2} : \tau$. Technically, we then have a case for each rule that could be used to prove this fact. In this case, there's only one, TypeAdd, so we can assume that this is the bottom rule in the derivation of $e : \tau$. Because we know this is the rule that was used, we know that $\tau = \mathsf{int}$ (we also get that $\overline{n_1}\mathsf{int}$: and $\overline{n_2} : \mathsf{int}$, but we knew that already.

[2] There are only axioms, so there's never an induction hypothesis.

Proof. By induction on the derivation of $e : \tau$.

- TypeNum. Then $e = \overline{n}$. By ValNum, e val.
- TypeString. Then $e =$ "s". By ValString, e val.
- TypeAdd. Then $e = e_1 + e_2$ and $\tau = \mathsf{int}$ and $e_1 : \mathsf{int}$ and $e_2 : \mathsf{int}$. By induction, e_1 val or $e_1 \mapsto e_1'$ for some e_1'.
  - Suppose e_1 val. Then, by canonical forms, $e_1 = \overline{n_1}$ for some n_1. By the other induction hypothesis, e_2 val or $e_2 \mapsto e_2'$ for some e_2'.
    - Suppose e_2 val. Then, by canonical forms, $e_2 = \overline{n_2}$ for some n_2. So we have $e = \overline{n_1} + \overline{n_2}$. By StepAdd, $e \mapsto \overline{n_1 + n_2}$.
    - Suppose $e_2 \mapsto e_2'$. Then, by StepSearchAddRight, $e \mapsto \overline{n_1} + e_2'$.
  - Suppose $e_1 \mapsto e_1'$. Then, by StepSearchAddLeft, $e \mapsto e_1' + e_2$.
- TypeCat. Similar to TypeAdd.
- TypeLen. Then $e = |e_0|$ and $\tau = \mathsf{int}$ and $e_0 : \mathsf{string}$. By induction, e_0 val or $e_0 \mapsto e_0'$ for some e_0'.
  - e_0 val. Then, by canonical forms, $e_0 =$ "s" for some s and $e = |\text{“}s\text{”}|$. By StepLen, $e \mapsto \overline{|s|}$.
  - $e_0 \mapsto e_0'$. Then, by StepSearchLen, $e \mapsto |e_0'|$.

□