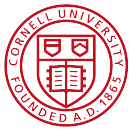
CS 4110 – Programming Languages and Logics
Lecture #19: Proving Type Safety for STLC



The end goal of adding types to the λ-calculus is *type soundness:* the guarantee that any well-typed program does not get stuck. In this lecture, we'll prove that the STLC's type system is sound.

While the STLC's type system is fairly simple, the proof will use a structure that we can re-use to prove soundness in much more complex languages. Every type soundness proof we'll do in the class will follow the same rough template—so to structure a new proof, start here and begin by writing down analogous *progress* and *preservation* lemmas.

1 Review: Simply-Typed Lambda Calculus

1.1 Syntax

expressions	$e ::= x \mid \lambda x{:}\tau.\, e \mid e_1\ e_2 \mid n \mid e_1 + e_2 \mid ()$
values	$v ::= \lambda x{:}\tau.\, e \mid n \mid ()$
types	$\tau ::= \textbf{int} \mid \textbf{unit} \mid \tau_1 \to \tau_2$

1.2 Dynamic Semantics

We use the small-step operational semantics that we defined using evaluation contexts.

$$E ::= [\cdot] \mid E\ e \mid v\ E \mid E + e \mid v + E$$

The small-step rules just ignore the type in abstractions.

$$\frac{e \to e'}{E[e] \to E[e']} \qquad \frac{}{(\lambda x{:}\tau.\, e)\ v \to e\{v/x\}} \qquad \frac{n = n_1 + n_2}{n_1 + n_2 \to n}$$

1.3 Static Semantics

Here are the rules for our typing judgment, $\Gamma \vdash e{:}\tau$.

$$\frac{}{\Gamma \vdash n{:}\textbf{int}}\ \text{T-Int} \qquad \frac{}{\Gamma \vdash ()\,{:}\,\textbf{unit}}\ \text{T-Unit} \qquad \frac{\Gamma \vdash e_1{:}\textbf{int} \quad \Gamma \vdash e_2{:}\textbf{int}}{\Gamma \vdash e_1 + e_2{:}\textbf{int}}\ \text{T-Add} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x{:}\tau}\ \text{T-Var}$$

$$\frac{\Gamma, x{:}\tau \vdash e{:}\tau'}{\Gamma \vdash \lambda x{:}\tau.\, e{:}\tau \to \tau'}\ \text{T-Abs} \qquad \frac{\Gamma \vdash e_1{:}\tau \to \tau' \quad \Gamma \vdash e_2{:}\tau}{\Gamma \vdash e_1\ e_2{:}\tau'}\ \text{T-App}$$

2 Type Soundness

Here's a formal statement of the type soundness property we want to the simply-typed λ-calculus:

Theorem (Type soundness)**.** *If* $\vdash e : \tau$ *and* $e \to^* e'$ *and* $e' \not\to$ *then* e' *is a value and* $\vdash e' : \tau$*.*

We will prove this theorem using two lemmas: *preservation* and *progress*. Intuitively, preservation says that if an expression e is well-typed, and e can take a step to e', then e' is well-typed. That is, evaluation preserves well-typedness. Progress says that if an expression e is well-typed, then either e is a value, or there is an e' such that e can take a step to e'. That is, well-typedness means that the expression cannot get stuck.

Together, these two lemmas suffice to prove type soundness. Given the preservation lemma, a trivial induction on the number of steps taken to reach e' from e establishes that $\vdash e' : \tau$. Then the progress lemma ensures that, if e' cannot take a step, then it must be a value.

Now we'll state and prove these two all-important lemmas.

2.1 Preservation

To prove preservation, we will need some extra tiny lemmas.

Lemma (Substitution)**.** *If* $x : \tau' \vdash e : \tau$ *and* $\vdash v : \tau'$ *then* $\vdash e\{v/x\} : \tau$*.*

Lemma (Context)**.** *If* $\vdash E[e] : \tau$ *and* $\vdash e : \tau'$ *and* $\vdash e' : \tau'$ *then* $\vdash E[e'] : \tau$*.*

We'll assume these without proof. (They're not difficult, but the proof of substitution can get rather long.) Equipped with these little lemmas, we're ready to move on to the main proof of preservation.

A quick note on proof strategy: to prove preservation, it's possible to induct either on the typing relation or on the small-step relation. Both have their advantages and disadvantages; we'll use the small-step relation here.

Lemma (Preservation)**.** *If* $\vdash e : \tau$ *and* $e \to e'$ *then* $\vdash e' : \tau$*.*

Proof. Assume $\vdash e : \tau$ and $e \to e'$. We need to show $\vdash e' : \tau$. We will do this by induction on the derivation of $e \to e'$.

- ADD

  Here, $e \equiv n_1 + n_2$, and $e' = n$ where $n = n_1 + n_2$, and $\tau = \textbf{int}$.

  By the typing rule T-INT, we have $\vdash e' : \textbf{int}$ as required.

- β-REDUCTION

  Here, $e \equiv (\lambda x : \tau'.\, e_1)\ v$ and $e' \equiv e_1\{v/x\}$.

  Since e is well-typed by assumption, we have derivations showing $\vdash \lambda x : \tau'.\, e_1 : \tau' \to \tau$ and $\vdash v : \tau'$. There is only one typing rule for abstractions, T-ABS, from which we know $x : \tau' \vdash e_1 : \tau$.

  By our substitution lemma above, we have $\vdash e_1\{v/x\} : \tau$ as required.

- CONTEXT

  Here, we have some context E such that $e = E[e_1]$ and $e' = E[e_2]$ for some e_1 and e_2 such that $e_1 \to e_2$.

  Since e is well-typed, we can show by induction on the structure of E that $\vdash e_1 : \tau_1$ for some τ_1. (This simple sub-induction is left as an exercise.)

  By the induction hypothesis and because we know $e_1 \to e_2$, we have $\vdash e_2 : \tau_1$. (Put intuitively, e_2 has the same type as the one we just established for e_1.)

  By our context lemma above, we have $\vdash E[e_2] : \tau$ as required.

□

2.2 Progress

To prove our progress lemma, we'll need one extra lemma that gives us the syntax forms for closed terms.

Lemma (Canonical Forms)**.** *If $\vdash v : \tau$, then*

1. *If τ is **int**, then v is a constant, i.e., some c.*
2. *If τ is $\tau_1 \to \tau_2$, then v is an abstraction, i.e., $\lambda x : \tau_1 . e$ for some x and e.*

Proof. The proof is by inspection of the typing rules.

i If τ is int, then the only rule which lets us give a value this type is T-INT.

ii If τ is $\tau_1 \to \tau_2$, then the only rule which lets us give a value this type is T-ABS.

□

Now we're ready to prove progress.

Lemma (Progress)**.** *If $\vdash e : \tau$ then either e is a value or there exists an e' such that $e \to e'$.*

Proof. We proceed by induction on the derivation of $\vdash e : \tau$.

- T-VAR

  This case is impossible, since a variable is not well-typed in the empty environment.

- T-UNIT, T-INT, T-ABS

  In all of these cases, e is a value.

- T-ADD

  Here $e \equiv e_1 + e_2$ and $\vdash e_1 : \textbf{int}$ and $\vdash e_2 : \textbf{int}$. By the inductive hypothesis, for $i \in \{1, 2\}$ (i.e., for both e_1 and e_2), either e_i is a value or there is an e_i' such that $e_i \to e_i'$.

  If e_1 is not a value, we have from above that $e_1 \to e_1'$. Therefore, by the CONTEXT rule, $e_1 + e_2 \to e_1' + e_2$.

  Otherwise, e_1 is a value. If e_2 is not a value, then by CONTEXT again, $e_1 + e_2 \to e_1 + e_2'$.

  Otherwise, both e_1 and e_2 are values. By our canonical forms lemma, $e_1 = n_1$ and $e_2 = n_2$ are both integer literals. By the ADD rule, we have $e_1 + e_2 \to n$ where $n = n_1 + n_2$.

- T-APP

  Here $e \equiv e_1\ e_2$ and $\vdash e_1 : \tau' \to \tau$ and $\vdash e_2 : \tau'$. By the inductive hypothesis, for $i \in \{1, 2\}$, either e_i is a value or there is an e_i' such that $e_i \to e_i'$.

  If e_1 is not a value, then by the above and by applying the CONTEXT rule, $e_1\ e_2 \to e_1'\ e_2$.

  Otherwise, e_1 is a value. If e_2 is not a value, then by CONTEXT, $e_1\ e_2 \to e_1\ e_2'$.

  If e_1 and e_2 are values, then, by our canonical forms lemma, e_1 is an abstraction $\lambda x : \tau'.\, e'$. Therefore, by β-REDUCTION, we have $e_1\ e_2 \to e'\{e_2/x\}$.

□

3 Type "Completeness"?

Not all expressions in the untyped lambda calculus are well-typed. Type soundness implies that any lambda calculus program that gets stuck is not well-typed.

But are there programs that *do not* get stuck that are not well-typed? In other words, does our type system unjustly rule out legal programs?

Unfortunately, the answer is yes. In particular, because the simply-typed lambda calculus requires us to specify a type for function arguments, any given function can only take arguments of one type. Consider, for example, the identity function $\lambda x.\, x$. This function may be applied to any argument, and it will not get stuck. However, we must provide a type for the argument. If we specify $\lambda x : \textbf{int}.\, x$, then this function can only accept integers, and the program $(\lambda x : \textbf{int}.\, x)\ ()$ is not well-typed, even though it does not get stuck. Indeed, in the simply-typed lambda calculus, there is a different identity function for each type.