CIS 7000-1 Homework 1

September 22, 2025

1 Analyzing type systems

Each of the following subsections of this problem describes a variant of STLC, including a (potentially) modified grammar, small-step operational semantics, and type system. Each of these variants is independent and you should consider them separately from all others.

For each variant, determine whether type safety holds, where type safety is defined to be the following property.

Definition 1.1 (Stuck)**.** A term e is *stuck* if it is not a value and there does not exists any e' such that $e \rightsquigarrow e'$.

Theorem 1.1 (Type safety)**.** If $\emptyset \vdash e \in \tau$ then for all e', such that $e \rightsquigarrow^* e'$, e' is not stuck.

If the type safety property fails, in a sentence or two, explain in English the source of the error and intuitively how a well-typed program can get stuck.

Regardless of whether type safety holds, state whether the properties of *substitution*, *preservation* and *progress* are true for that system, as stated in the lecture notes. For each false property, give a concrete counter-example and clearly explain why it is a counter-example.

For example, consider the preservation property: If $\emptyset \vdash e \in \tau$ and $e \rightsquigarrow e'$ then $\emptyset \vdash e' \in \tau$.

To give a counter-example, supply a specific e, a specific e' and a specific τ. Then explain why the preservation statement is false for the specific terms you have supplied. To do that, you will show a derivation of $\emptyset \vdash e \in \tau$ to demonstrate e is well-typed and a derivation to show that $e \rightsquigarrow e'$. Then explain why no derivation of $\emptyset \vdash e' \in \tau$ exists (eg: show a partial derivation and explain why you get stuck finishing it off with the rules supplied.)

1.1 Null

Suppose we add a new value called **null**. As in most programming languages, we also add the following typing rule so that **null** has any type:

$$\frac{}{\Gamma \vdash \mathbf{null} \in \tau} \ \text{T\_NULL}$$

1. Is STLC with this modification type safe?

   No. This modification is NOT type safe. The error occurs from the fact that **null** can be any type. Thus function applications can get stuck if the function itself is null.

   Consider if we do **null 5**. By rule T-NULL, we can assign **null** : **Nat** $\to \tau$ for any τ. By T-LIT, $\emptyset \vdash 5 \in$ **Nat**. Hence $\emptyset \vdash$ **null 5** $\in \tau$. But operational semantics has no reduction rule for applying null to an argument. So **null 5** is not a value and cannot step. It is stuck, violating type safety.

2. Does *substitution* hold?

   Substition holds. Adding **null** doesn't affect substitution since **null** itself never binds variables.

3. Does *preservation* hold?

   Preservation holds because **null** we haven't added any new small-step rules.

4. Does *progress* hold?

   Progress does not hold. The problem occurs from the fact that **null** can be any type. Thus function applications can get stuck if the function itself is null.

   Consider if we do **null 5**. In this case, $\emptyset \vdash \textbf{null} \in \textbf{Nat} \to \tau$. This function application is now stuck and cannot be further simplified.

1.2 Void

Suppose we add a new type to STLC called **Void**. But that is it. We don't add any new terms, typing rules or small-step reduction rules. This type is called **Void** because it is empty; there are no closed values with this type.

1. Is STLC with this modification type safe?

   Yes, it is still type safe. Intuitively, no closed values can have this type and we have not added any new terms with this type or any new rules that will change the semantics of our existing language so STLC with Void is still type safe.

2. Does *substitution* hold?

   Yes. No new rules/terms were added, so substitution is the same as in STLC.

3. Does *preservation* hold?

   Yes. No new rules/terms were added, so preservation is the same as in STLC.

4. Does *progress* hold?

   Yes. No new rules/terms were added, so progress is the same as in STLC.

1.3 A mystery language

Suppose we add the following new rules to STLC, where Σ is some fixed map from natural numbers to types. This map is defined for all numbers, but can return any type.

$$\frac{\Sigma(k) = \tau_1 \to \tau_2}{\Gamma \vdash k \in \tau_1 \to \tau_2} \quad \text{T\_ARR\_PTR}$$

$$\frac{}{k\ v \rightsquigarrow (\lambda x.k\ x)\ v} \quad \text{S\_APP\_NAT}$$

1. Is STLC with this modification type safe?

   Yes. The new small-step reduction rule ensures that any $k\ v$ can be evaluated without getting stuck.

2. Does *substitution* hold?

   Yes, substitution holds. None of the new rules bind variables.

3. Does *preservation* hold?

   Yes. Our new S-APP-NAT rule still preserves the types given in the map.

4. Does *progress* hold?

   Yes. $k\ v$ will always step to a lambda application which we can step normally with STLC rules.

1.4 STLC–

Suppose we remove the typing rule for natural numbers, rule T-LIT, from STLC.

1. Is STLC with this modification type safe?

   This modification is type-safe. It significantly limits the expressions that typecheck in our language, but any expression that does typecheck will still evaluate without getting stuck.

2. Does *substitution* hold?

   Yes, substitution holds as in STLC for expressions that typecheck.

3. Does *preservation* hold?

   Yes, preservation holds as in STLC for expressions that typecheck.

4. Does *progress* hold?

   Yes, progress holds as in STLC for expressions that typecheck.

1.5 STLC with lists

Suppose we add lists to STLC by adding two new expression forms, $\textbf{cons}\, e_1\, e_2$ and **nil**. These new forms are both values.

$$\begin{array}{l} \tau ::= \ldots \mid \textbf{List}\, \tau \\ v ::= \ldots \mid \textbf{cons}\, e_1\, e_2 \mid \textbf{nil} \\ e ::= \ldots \mid \textbf{cons}\, e_1\, e_2 \mid \textbf{nil} \end{array}$$

The typing rules for lists allows us to construct any sort of list out of **nil** and **cons**.

$$\frac{\Gamma \vdash e_1 \in \tau \quad \Gamma \vdash e_2 \in \textbf{List}\, \tau}{\Gamma \vdash \textbf{cons}\, e_1\, e_2 \in \textbf{List}\, \tau}\ \text{T\_CONS}$$

$$\frac{}{\Gamma \vdash \textbf{nil} \in \textbf{List}\, \tau}\ \text{T\_NIL}$$

We also will allow programmers to access the elements of a list through projection. We will reuse the syntax of function application for list projection: if the first argument is some list l and the second argument is some number k, then the application looks up the kth element of the list l:

$$\frac{\Gamma \vdash e_1 \in \textbf{List}\, \tau \quad \Gamma \vdash e_2 \in \textbf{Nat}}{\Gamma \vdash e_1\ e_2 \in \tau}\ \text{T\_NTH}$$

$$\frac{}{(\textbf{cons}\, v_1\, v_2)\ 0 \rightsquigarrow v_1}\ \text{S\_APP\_ZERO}$$

$$\frac{}{(\textbf{cons}\, v_1\, v_2)\ (S\, k) \rightsquigarrow v_2\ k}\ \text{S\_APP\_SUCC}$$

1. Is STLC with this modification type safe?

   This modification is not type-safe. There is no rule for evaluating applications of the form $\textbf{nil}\ k$, even though they typecheck.

2. Does *substitution* hold?

   Yes, substitution holds. None of the new rules bind variables.

3. Does *preservation* hold?

   Yes, preservation holds. The new step rules we added preserve typing.

4. Does *progress* hold?

   Progress doesn't hold. Applications of the form **nil** k do not step to anything and they are also not values.

1.6 Simply-typed function pointers

Suppose we modify STLC to use *function pointers* instead of anonymous functions. To do so, we assume the existence of μ, a fixed map from natural numbers to abstractions and Σ, a map from natural numbers to types.

We also remove the rules that type check and step anonymous functions (as they can no longer appear directly in programs), rule T-ABS and rule S-BETA, and replace them with the following two rules that allow natural numbers to used as function pointers.

$$\frac{\Sigma(k) = \tau_1 \to \tau_2}{\Gamma \vdash k \in \tau_1 \to \tau_2} \quad \text{T\_ARR\_PTR}$$

$$\frac{\mu(k) = \lambda x.e}{k\ v \rightsquigarrow e[v/x]} \quad \text{S\_APP\_PTR}$$

We also assume that all functions stored in the table typecheck according to this type system:

Assumption 1.1 (Table typing)**.** For all k, if $\mu(k) = \lambda x.e$ and $\Sigma(k) = \tau_1 \to \tau_2$ then $x{:}\tau_1 \vdash e \in \tau_2$.

1. Is STLC with this modification type safe?

   This language modification preserves type safety. Function applications with function pointers that typecheck eventually step the same way as anonymous functions in STLC.

2. Does *substitution* hold?

   Yes, substitution holds. We haven't introduced new variable binding rules.

3. Does *preservation* hold?

   Yes, preservation holds. The S-APP-PTR rule preserves typing.

4. Does *progress* hold?

   Yes, progress holds. Any term $k\ v$ steps to $e[v/x]$ which steps as in normal STLC.

2 Preservation and Progress proofs

The next part of the homework assignment involves completing the proofs of preservation and progress for two extensions of STLC. If you would like to use Rocq to mechanize these proofs, you can find initial code in the 'homework' directory of the course repository.

2.1 Let binding

Consider adding let expressions to STLC. To do so we extend the grammar, type system, and operational semantics as follows. We add a new expression form that binds the variable x in the body of the let expression

e_2.

$$e ::= \ldots \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2$$

We add a single new typing rule:

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \in \tau_1 \\ \Gamma, x : \tau_1 \vdash e_2 \in \tau_2\end{array}}{\Gamma \vdash \textbf{let}\ x\ = e_1\ \textbf{in}\ e_2 \in \tau_2}\ \text{T\_LET}$$

and two new evaluation rules:

$$\frac{}{\textbf{let}\ x\ = v\ \textbf{in}\ e \rightsquigarrow e[v/x]}\ \text{S\_LETV}$$

$$\frac{e_1 \rightsquigarrow e_1'}{\textbf{let}\ x\ = e_1\ \textbf{in}\ e_2 \rightsquigarrow \textbf{let}\ x\ = e_1'\ \textbf{in}\ e_2}\ \text{S\_LET\_CONG}$$

1. Extend the preservation proof. This proof is by induction on evaluation steps. That means there will need to be two new cases for rules S-LETV and S-LET-CONG.
2. Extend the progress proof. This proof is by induction on the typing judgement. That means there will be one new case for rule T-LET.

Lemma 2.1 (Preservation)**.** If $\emptyset \vdash e \in \tau$ and $e \rightsquigarrow e'$ then $\emptyset \vdash e' \in \tau$.

Proof. The proof is by induction on the derivation of the reduction. There are cases for each of the rules that could have been used to conclude $e \rightsquigarrow e'$.

- In the case of rule S-LETV, we want to prove that if $\emptyset \vdash\ \textbf{let}\ x = v\ \textbf{in}\ e \in \tau$ then $\emptyset \vdash e[v/x] \in \tau$. We will assume that substitution has already been extended to let expressions.

  The inversion of T-LET tells us that $\emptyset \vdash v \in \tau_1$ and $x : \tau_1 \vdash e \in \tau$ so if we apply substitution, we get exactly that $\emptyset \vdash e[v/x] \in \tau$.

- In the case of rule S-LET-CONG, we want to prove that if

  $$\emptyset \vdash\ \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \in \tau$$

  and

  $$\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \rightsquigarrow \textbf{let}\ x = e_1'\ \textbf{in}\ e_2$$

  then

  $$\emptyset \vdash\ \textbf{let}\ x = e_1'\ \textbf{in}\ e_2 \in \tau$$

  The inversion of T-LET on the premise tells us that $\emptyset \vdash e_1 \in \tau_1$ and $x : \tau_1 \vdash e_2 \in \tau$.

  If we apply our typing rule T-LET to the goal, we can break our proof down into two parts.

  - $\emptyset \vdash e_1' \in \tau_1$: The induction hypothesis tells us that since $e_1 \rightsquigarrow e_1'$ and $\emptyset \vdash e_1 \in \tau_1$, $\emptyset \vdash e_1' \in \tau_1$.
  - $x : \tau_1 \vdash e_2 \in \tau$: This was previously proven through inversion.

So we have proven both sub-cases and the proof is complete. □

Lemma 2.2 (Progress)**.** If $\emptyset \vdash e \in \tau$ then either e is a value or there exists an e' such that $e \rightsquigarrow e'$.

Proof. We prove this lemma by induction in the typing derivation. In the rules where e is already a value, then the proof is trivial. Otherwise, we need to extend the proof to T-LET. We have $\emptyset \vdash\ \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \in \tau$ The typing rule tells us that $\emptyset \vdash e_1 \in \tau_1$, so by IH, we know either

- $e_1 = v$ is a value. In this case, we see that our expression steps to $e_2[v/e_1]$ by S-LETV and we are done.
- e steps: $\exists e'$ such that $e_1 \rightsquigarrow e_1'$. In this case, we see that our expression steps to **let** $x = e_1'$ **in** e_2 by S-LET-CONG and we are done as well.

□

2.2 Natural number recursion

Proof preservation and progress for the extension of STLC with a successor and primitive recursion operation as described in the lecture notes.

$$e ::= \ldots \mid \mathbf{succ}\, e \mid \mathbf{nrec}\, e\, \mathbf{of}\, \{0 \Rightarrow e_1;\, \mathbf{S}\, x \Rightarrow e_2\}$$

Lemma 2.3 (Preservation)**.** If $\emptyset \vdash e \in \tau$ and $e \rightsquigarrow e'$ then $\emptyset \vdash e' \in \tau$.

Proof. The proof is by induction on the derivation of the reduction. There are cases for each of the rules that could have been used to conclude $e \rightsquigarrow e'$.

- S-SUCC-LIT. This proof is trivial. $\mathbf{succ}\ k \rightsquigarrow S\ k$ and $\emptyset \vdash \mathbf{succ}\ k \in \mathbf{Nat}$. We want to prove that $\emptyset \vdash S\ k \in \mathbf{Nat}$, which is true by T-LIT.
- S-SUCC-CONG. Here $\mathbf{succ}\ e \rightsquigarrow \mathbf{succ}\ e'$ and $e \rightsquigarrow e'$. We know from T-SUCC that $\emptyset \vdash e \in \mathbf{Nat}$. We want to prove that $\emptyset \vdash \mathbf{succ}\ e' \in \mathbf{Nat}$.

  If we apply T-SUCC then we know the goal is just to prove that $\emptyset \vdash e' \in \mathbf{Nat}$. The induction hypothesis tells us that since $\emptyset \vdash e \in \mathbf{Nat}$ and $e \rightsquigarrow e'$, then $\emptyset \vdash e' \in \mathbf{Nat}$ and we are done.
- S-NREC-ZERO. Here we have $\emptyset \vdash \mathbf{nrec}\ 0\ \mathbf{of}\ \{0 \Rightarrow e_1; S\ x \Rightarrow e_2\} \in \tau$ and we merely want to prove that $\emptyset \vdash e_1 \in \tau$. This is trivially true because of the inversion of the typing rule T-NREC.
- S-NREC-SUCC. Here we have $\emptyset \vdash \mathbf{nrec}\ (S\ k)\ \mathbf{of}\ \{0 \Rightarrow e_1; S\ x \Rightarrow e_2\} \in \tau$ and we want to prove that $\emptyset \vdash (e_2[k/x])\ (\mathbf{nrec}\ k\ \mathbf{of}\ \{0 \Rightarrow e_1; S\ x \Rightarrow e_2\}) \in \tau$.

  We apply inversion of T-NREC on the premise and we find that $\emptyset \vdash S\ k \in \mathbf{Nat}$, $\emptyset \vdash e_1 \in \tau$, and $x : \mathbf{Nat} \vdash e_2 \in \tau \to \tau$.

  We first apply T-APP and we now have to prove two things:

  - $\emptyset \vdash e_2[k/x] \in \tau \to \tau$. Here we apply substitution. Introducing k changes the context and our goal becomes $k : \mathbf{Nat} \vdash e_2 \in \tau \to \tau$ which we know from the inversion of T-NREC on the premise.
  - $\emptyset \vdash (\mathbf{nrec}\ k\ \mathbf{of}\ \{0 \Rightarrow e_1; S\ x \Rightarrow e_2\}) \in \tau$. If we apply T-NREC on the goal we split our goal into three parts:
    - $\emptyset \vdash S\ k \in \mathbf{Nat}$. This is trivially true through T-LIT.
    - $\emptyset \vdash e_1 \in \tau$. This is also trivially true through inversion of T-NREC on the premise.
    - $x : \mathbf{Nat} \vdash e_2 \in \tau \to \tau$. This is also trivially true through inversion of T-NREC on the premise.
- S-NREC-CONG. Here we have $\emptyset \vdash \mathbf{nrec}\ e\ \mathbf{of}\ \{0 \Rightarrow e_1; S\ x \Rightarrow e_2\} \in \tau$ and $e \rightsquigarrow e'$. We want to prove that $\emptyset \vdash \mathbf{nrec}\ e'\ \mathbf{of}\ \{0 \Rightarrow e_1; S\ x \Rightarrow e_2\} \in \tau$.

  We apply inversion of T-NREC on the premise and we find that $\emptyset \vdash e \in \mathbf{Nat}$, $\emptyset \vdash e_1 \in \tau$, and $x : \mathbf{Nat} \vdash e_2 \in \tau \to \tau$.

  If we now apply T-NREC on the goal, we split our goal into three parts:

  - $\emptyset \vdash e' \in \mathbf{Nat}$. This is covered by induction hypothesis because $\emptyset \vdash e \in \mathbf{Nat}$, and $e \rightsquigarrow e'$.
  - $\emptyset \vdash e_1 \in \tau$. This is trivially true through inversion of T-NREC on the premise.
  - $x : \mathbf{Nat} \vdash e_2 \in \tau \to \tau$. This is also trivially true through inversion of T-NREC on the premise.

□

Lemma 2.4 (Progress)**.** If $\emptyset \vdash e \in \tau$ then either e is a value or there exists an e' such that $e \rightsquigarrow e'$.

Proof. We prove this lemma by induction in the typing derivation. We need to extend the proof to the two new typing derivations.

- T-SUCC. $\emptyset \vdash \mathbf{succ}\ e \in \mathbf{Nat}$. We want to prove that $\mathbf{succ}\ e$ is either a value or it steps. Through inversion of T-SUCC on the premise we know that $\emptyset \vdash e \in \mathbf{Nat}$. By induction hypothesis we know that either
  - e is a value. If e is a value then by canonical forms we know that $e = k$ because $\emptyset \vdash e \in \mathbf{Nat}$. Thus $\mathbf{succ}\ k \rightsquigarrow S\ k$. We know this through S-SUCC-LIT.
  - e steps: $\exists e'$ such that $e \rightsquigarrow e'$. Here we know that $\mathbf{succ}\ e \rightsquigarrow \mathbf{succ}\ e'$ simply through S-SUCC-CONG.
- T-NREC. $\emptyset \vdash \mathbf{nrec}\ e\ \mathbf{of}\ \{0 \Rightarrow e_1; S\ x \Rightarrow e_2\} \in \tau$. We want to prove that this is either a value or it steps.

  Through inversion of T-NREC on the premise we know that $\emptyset \vdash e \in \mathbf{Nat}$, $\emptyset \vdash e_1 \in \tau$, and $x : \mathbf{Nat} \vdash e_2 \in \tau \to \tau$.

  By induction hypothesis we know that either
  - e is a value. If e is a value then by canonical forms we know that $e = k$ because $\emptyset \vdash e \in \mathbf{Nat}$ so either
    - $k = 0$, in which case we know $\mathbf{nrec}\ 0\ \mathbf{of}\ \{0 \Rightarrow e_1; S\ x \Rightarrow e_2\} \rightsquigarrow e_1$ through S-NREC-ZERO.
    - $k = S\ k'$ in which case we know

      $$\mathbf{nrec}\ S\ k\ \mathbf{of}\ \{0 \Rightarrow e_1; S\ x \Rightarrow e_2\} \rightsquigarrow (e_2[k/x])\ (\mathbf{nrec}\ k\ \mathbf{of}\ \{0 \Rightarrow e_1; S\ x \Rightarrow e_2\})$$

      through S-NREC-SUCC.
  - e steps: $\exists e'$ such that $e \rightsquigarrow e'$. Here we know that

    $$\mathbf{nrec}\ S\ k\ \mathbf{of}\ \{0 \Rightarrow e_1; e \Rightarrow e_2\} \rightsquigarrow \mathbf{nrec}\ e'\ \mathbf{of}\ \{0 \Rightarrow e_1; S\ x \Rightarrow e_2\}$$

    through S-NREC-CONG.

□