CS345H: Programming Languages

Lecture 10: Basic Type Checking

Thomas Dillig

Outline

- We will write type systems for multiple languages
- We will formally see how to define soundness
- We will learn how to prove soundness of a type system

The let language

- Recall from last time the following small language (let language):

$$\begin{array}{lcl} S & \rightarrow & \text{integer} \mid \text{string} \mid \text{identifier} \\ & & \mid S_1 + S_2 \mid S_1 :: S_2 \\ & & \mid \text{let } id : \tau \ = \ S_1 \text{ in } S_2 \\ \tau & \rightarrow & Int \mid String \end{array}$$

- Here are again its operational semantics:

$$\frac{\text{integer } i}{E \vdash i : i} \qquad \frac{\text{string } s}{E \vdash s : s} \qquad \frac{\text{identifier } id}{E \vdash id : E(id)} \qquad \frac{\begin{array}{c} E \vdash S_1 : i_1 \\ E \vdash S_2 : i_2 \end{array}}{E \vdash S_1 + S_2 : i_1 + i_2}$$

$$\frac{\begin{array}{c} E \vdash S_1 : s_1 \\ E \vdash S_2 : s_2 \end{array}}{E \vdash S_1 :: S_2 : \text{concat}(s_1, s_2)} \qquad \frac{\begin{array}{c} E \vdash S_1 : e_1 \\ E[\text{id} \leftarrow e_1] \vdash S_2 : e_2 \end{array}}{E \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : e_2}$$

Type System

- We also saw last time how we can write typing rules that compute the type of an expression.

$$\frac{\text{integer } i}{\Gamma \vdash i : Int} \qquad \frac{\text{string } s}{\Gamma \vdash s : String} \qquad \frac{\text{identifier } id}{\Gamma \vdash id : \Gamma(id)}$$

$$\frac{\begin{array}{c} \Gamma \vdash S_1 : Int \\ \Gamma \vdash S_2 : Int \end{array}}{\Gamma \vdash S_1 + S_2 : Int} \qquad \frac{\begin{array}{c} \Gamma \vdash S_1 : String \\ \Gamma \vdash S_2 : String \end{array}}{\Gamma \vdash S_1 :: S_2 : String}$$

$$\frac{\begin{array}{l} \Gamma \vdash S_1 : \tau_1 \\ \tau = \tau_1 \\ \Gamma[\text{id} \leftarrow \tau] \vdash S_2 : \tau_3 \end{array}}{\Gamma \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : \tau_3}$$

Correspondence between Concrete and Abstract Semantics

- Observe that there is a strong relationship between the operational semantics (concrete semantics) and the typing rules (abstract semantics)
  - The concrete environment E corresponds to the abstract type environment Γ
  - The structure of the abstract and concrete rules are analogous
- Key Difference: Concrete semantics compute a particular value, while abstract semantics compute a type

Some Notation

- We write $\gamma(\tau)$ for the concretization of the abstract value τ. We call γ the concretization function
- Example: $\gamma(Int) = \{\ldots, -1, 0, 1, 2, 3, \ldots\}$
- We write $\alpha(v)$ for the abstraction of the concrete value v. We call α the abstraction function
- Example: $\alpha(42) = Int$
- Definition: An abstraction is a Galois Connection if $\alpha(\gamma(\tau)) = \tau$ for all abstract values τ
- Question: Is our abstract domain of types a Galois connection? Yes, $\alpha(\gamma(Int)) = Int$ and $\alpha(\gamma(String)) = String$

Galois Connection

- Galois connection means that if we want to relate a concrete value v and abstract value τ, the following are equivalent:
- $\alpha(v) = \tau$
- $v \in \gamma(\tau)$
- Think of it as a well-formed abstraction
- In this class, we are only interested in Galois connections

Soundness

- For out type system to be sound, we require that for any program, the concrete value v of this program is compatible with the type τ computed.
- Formally, we state this property as follows:
- If $E \vdash e : v$ and $\Gamma \vdash e : \tau$, then $v \in \gamma(\tau)$
- This means that the type we give to every expression always overapproximates the type of the concrete value
- We can safely rely on the static types computed
- Slogan: "Well-typed programs cannot go wrong"

Soundness Cont.

- Clearly, not every type system is sound or useful to prevent run-time errors
- Therefore, we need a way to prove that a type system we design is actually sound and useful.
- There are many ways of proving correspondence between abstract and concrete semantics, but the most popular strategy for types is to split the problem into two:
  1. Preservation: Soundness is preserved under transition rules
  2. Progress: A well-typed program never "gets stuck" when executing operational semantics (no run-time errors).
- Preservation states that your type system is an overapproximation while progress states that your type system is expressive enough to prevent all run-time errors

How to Prove Preservation

- Preservation: If $E \vdash e : v$ and $\Gamma \vdash e : \tau$, then $v \in \Gamma(\tau)$ (or equivalently $\alpha(v) = \tau$)
- Preservation must be argued inductively, specifically via structural induction on the program expressions
  - We first need to argue preservation for all the base cases: Base case: rules with no $\vdash$ in their hypotheses
  - Then, for the inductive rules, we assume that preservation holds for all subexpressions and prove that it holds for the current expression.
- This is a very powerful proof technique!

Proving Preservation

- Let's prove preservation of our type system, first without identifiers and let bindings:
- Base case 1: $\dfrac{\text{integer } i}{E \vdash i : i} \quad \dfrac{\text{integer } i}{\Gamma \vdash i : Int}$

  Need to prove that $\alpha(i) = Int$
  $\Rightarrow$ This follows directly from the hypothesis that i is an integer

Proving Preservation

- Base case 2: $\dfrac{\text{string } s}{E \vdash s : s} \quad \dfrac{\text{string } s}{\Gamma \vdash s : String}$

  Also follows immediately that $\alpha(s) = String$

Proving Preservation

- Inductive case 1:

$$\frac{E \vdash S_1 : i_1 \quad E \vdash S_2 : i_2}{E \vdash S_1 + S_2 : i_1 + i_2} \qquad \frac{\Gamma \vdash S_1 : Int \quad \Gamma \vdash S_2 : Int}{\Gamma \vdash S_1 + S_2 : Int}$$

- By the inductive hypothesis we know that $\alpha(i_1) = Int$ and $\alpha(i_2) = Int$. Since the value $i_1 + i_2$ is also an integer, $\alpha(i_1 + i_2) = Int$

Proving Preservation

- Inductive case 2:

$$\frac{E \vdash S_1 : s_1 \quad E \vdash S_2 : s_2}{E \vdash S_1 :: S_2 : \text{concat}(s_1, s_2)} \qquad \frac{\Gamma \vdash S_1 : String \quad \Gamma \vdash S_2 : String}{\Gamma \vdash S_1 :: S_2 : String}$$

- By the inductive hypothesis we know that $\alpha(s_1) = String$ and $\alpha(s_2) = String$. Since the value $concat(s_1, s_2)$ is also a string, $\alpha(concat(s_1, s_2)) = String$

Proving Preservation with Identifiers

- But what about the two rules involving identifiers?

$$\frac{\text{identifier } id}{E \vdash id : E(id)} \qquad \frac{\text{identifier } id}{\Gamma \vdash id : \Gamma(id)}$$

$$\frac{E \vdash S_1 : e_1 \quad E[\text{id} \leftarrow e_1] \vdash S_2 : e_2}{E \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : e_2} \qquad \frac{\Gamma \vdash S_1 : \tau_1 \quad \tau = \tau_1 \quad \Gamma[\text{id} \leftarrow \tau] \vdash S_2 : \tau_3}{\Gamma \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : \tau_3}$$

- To prove the base case, we need to know that the values in Γ and E agree.
- Definition: Concrete environment E and abstract environment Γ agree if for any identifier x $\Gamma(x) = \alpha(E(x))$, written as $\Gamma \sim E$
- Therefore, we first need to prove agreement before showing the preservation of the identifier rules

Proving Agreement

- Fortunately, proving agreement of E and Γ is easy, again by induction, on the number of mappings in E and Γ
- Base case: E and Γ are empty: $\Rightarrow$ they trivially agree
- Clearly, rules that do not change E or Γ cannot break agreement.
- Therefore, we only have to prove agreement for the following rule:

$$\frac{E \vdash S_1 : e_1 \quad E[\text{id} \leftarrow e_1] \vdash S_2 : e_2}{E \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : e_2} \qquad \frac{\Gamma \vdash S_1 : \tau_1 \quad \tau = \tau_1 \quad \Gamma[\text{id} \leftarrow \tau] \vdash S_2 : \tau_3}{\Gamma \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : \tau_3}$$

Proving Agreement

$$\frac{E \vdash S_1 : e_1 \quad E[\text{id} \leftarrow e_1] \vdash S_2 : e_2}{E \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : e_2} \qquad \frac{\Gamma \vdash S_1 : \tau_1 \quad \tau = \tau_1 \quad \Gamma[\text{id} \leftarrow \tau] \vdash S_2 : \tau_3}{\Gamma \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : \tau_3}$$

- Here, assuming preservation, we know that $\alpha(e_1) = \tau$. By the inductive hypothesis, we also know that $\Gamma \sim E$.
- Therefore, we also know that $\Gamma[id \leftarrow \tau] \sim E[id \leftarrow e_1]$
- Important: We proved agreement in the inductive case assuming preservation!

Proving Preservation with Identifiers

- Now, we can assume agreement when proving preservation:
- Base case:

$$\frac{\text{identifier } id}{E \vdash id : E(id)} \qquad \frac{\text{identifier } id}{\Gamma \vdash id : \Gamma(id)}$$

- This follows immediately since by our assumption $\Gamma \sim E$

Proving Preservation with Identifiers cont.

- Inductive case:

$$\frac{E \vdash S_1 : e_1 \quad E[\text{id} \leftarrow e_1] \vdash S_2 : e_2}{E \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : e_2} \qquad \frac{\Gamma \vdash S_1 : \tau_1 \quad \tau = \tau_1 \quad \Gamma[\text{id} \leftarrow \tau] \vdash S_2 : \tau_3}{\Gamma \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : \tau_3}$$

- By the inductive hypothesis, we know that $\alpha(e_2) = \tau_3$. This is what we want to prove.
- Observe: We combined agreement and preservation for this proof to work.
  - The preservation proof works assuming that agreement holds
  - The agreement proof works assuming that preservation holds
- As long as both properties hold initially, this is fine!

On to Progress

- We have now shown preservation of our type system
- Intuitively: We now know that that abstract value we compute will always overapproximate the concrete value for any program
- Now, we want to prove that our type system is powerful enough to prevent run-time type errors
- Or more formally, our operational semantics never "get stuck"
- Progress: We need to prove that every program that can be typed under our typing rules will not not "get stuck" in the operational semantics
- Progress is a very strong property that few real type systems obey!

Proving Progress

- We will again prove progress by structural induction:
  - Base case 1: Integer

$$\frac{\text{integer } i}{E \vdash i : i} \qquad \frac{\text{integer } i}{\Gamma \vdash i : Int}$$

Clearly, if i types as an integer, the corresponding operational semantics rule applies

  - Base case 2: String

$$\frac{\text{string } s}{E \vdash s : s} \qquad \frac{\text{string } s}{\Gamma \vdash s : String}$$

Clearly, if s types as a string, the corresponding operational semantics rule applies

Proving Progress

- Base case 3: Identifier

$$\frac{\text{identifier } id}{E \vdash id : E(id)} \qquad \frac{\text{identifier } id}{\Gamma \vdash id : \Gamma(id)}$$

Assuming agreement, we know that if the mapping $id \mapsto \tau$ is present in Γ, the mapping $id \mapsto v$ is present in E. Since this is the only hypothesis (precondition) of the operational semantics rule, it must therefore always apply in all well-types programs

Proving Progress

- Inductive case 1:

$$\frac{E \vdash S_1 : i_1 \quad E \vdash S_2 : i_2}{E \vdash S_1 + S_2 : i_1 + i_2} \qquad \frac{\Gamma \vdash S_1 : Int \quad \Gamma \vdash S_2 : Int}{\Gamma \vdash S_1 + S_2 : Int}$$

We know from the inductive hypothesis that the evaluation of S_1 and S_2 will never get stuck. We also know from preservation that the expressions S_1 and S_2 must evaluate to integers if they are typed Int, therefore the operational semantics rule for plus will always apply since the hypotheses only require that i_1 and i_2 are integers

Proving Progress

- Inductive case 2:

$$\frac{E \vdash S_1 : s_1 \quad E \vdash S_2 : s_2}{E \vdash S_1 :: S_2 : \text{concat}(s_1, s_2)} \qquad \frac{\Gamma \vdash S_1 : String \quad \Gamma \vdash S_2 : String}{\Gamma \vdash S_1 :: S_2 : String}$$

We know from the inductive hypothesis that the evaluation of S_1 and S_2 will never get stuck. We also know from preservation that the expressions S_1 and S_2 must evaluate to strings if they are typed String, therefore the operational semantics rule for concatenation will always apply since the hypotheses only require that s_1 and s_2 are strings

Proving Progress

- Inductive case 3:

$$\frac{E \vdash S_1 : e_1 \quad E[\text{id} \leftarrow e_1] \vdash S_2 : e_2}{E \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : e_2} \qquad \frac{\Gamma \vdash S_1 : \tau_1 \quad \tau = \tau_1 \quad \Gamma[\text{id} \leftarrow \tau] \vdash S_2 : \tau_3}{\Gamma \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : \tau_3}$$

Here, we know from the inductive hypothesis that $E \vdash S_1 : e_1$ and $E[\text{id} \leftarrow e_1] \vdash S_2 : e_2$ will not get stuck since they are well-typed. Therefore, this rule will also always apply.

Preservation + Progress

- We now proved both preservation and progress of our small type system on the let language.
- Important Point: You can only prove progress and preservation of a type system with respect to an operational semantics
- Poofs of preservation and progress are always by structural induction
- If you have an environment, you usually need to show agreement to prove preservation
- These proofs tend to always follow the same pattern, so follow this strategy on homeworks/exams

Adding the Lambda to our language

- Let us add the lambda construct to the let-language. I will call this the lambda-language:

$$\begin{aligned} S \;\rightarrow\; & \text{integer} \mid \text{string} \mid \text{identifier} \\ & \mid S_1 + S_2 \mid S_1 :: S_2 \\ & \mid \text{let } id : \tau \;=\; S_1 \text{ in } S_2 \\ & \mid \lambda x : \tau . S_1 \\ & \mid (S_1 \; S_2) \\ \tau \;\rightarrow\; & Int \mid String \mid \tau_1 \rightarrow \tau_2 \end{aligned}$$

- The operational semantics of the new constructs are as follows:

$$\frac{}{E \vdash \lambda x : \tau . S_1 : \lambda x : \tau \; . S_1} \qquad \frac{E \vdash S_1 : \lambda x : \tau . e \quad E \vdash S_2 : e_2 \quad E \vdash e[e_2/x] : e_r}{E \vdash (S_1 \; S_2) : e_r}$$

Typing rules for lambda and Application

- Lambda:

$$\frac{\Gamma[x \leftarrow \tau_1] \vdash S_1 : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . S_1 : \tau_1 \rightarrow \tau_2}$$

- Application:

$$\frac{\Gamma \vdash S_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash S_2 : \tau_1}{\Gamma \vdash (S_1 \; S_2) : \tau_2}$$

- Observe that these almost exactly correspond to the operational semantics!
- But there is one difference: The body of the let is type checked at the definition, but only evaluated at the application

Preservation for lambda

- Lambda:

$$\frac{}{E \vdash \lambda x : \tau . S_1 : \lambda x : \tau \; . S_1} \qquad \frac{\Gamma[x \leftarrow \tau_1] \vdash S_1 : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . S_1 : \tau_1 \rightarrow \tau_2}$$

- First, we observe that if $\Gamma[x \leftarrow \tau_1] \vdash S_1 : \tau_2$ holds, we know by our inductive hypothesis that $\alpha(E \vdash S_1[v/x]) = \tau_2$ for any value v of type τ_1. Therefore, the type of this rule is $\tau_1 \rightarrow \tau_2$

Preservation for Application

- Application:

$$\frac{E \vdash S_1 : \lambda x : \tau . e \quad E \vdash S_2 : e_2 \quad E \vdash e[e_2/x] : e_r}{E \vdash (S_1 \; S_2) : e_r} \qquad \frac{\Gamma \vdash S_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash S_2 : \tau_1}{\Gamma \vdash (S_1 \; S_2) : \tau_2}$$

- First, we observe by our inductive hypothesis that if the type of S_1 is $\tau_1 \rightarrow \tau_2$, the first hypothesis in the concrete rule must always apply. Second, by the inductive hypothesis we know that $\alpha(e_2) = \tau_1$. Since the type of S_1 is $\tau_1 \rightarrow \tau_2$, we can therefore safely conclude that $\alpha(e_r) = \tau_2$

Preservation Proof

- Question: Why could we not formulate the typing rules for lambda and application symmetric to the operational semantics?
- Answer: Because if we try to type check the body of a lambda at the application site, we have no way of knowing the name of the variable bound in this lambda statement
- This is typical: When typing functions, we usually always examine the function body before it is used

Progress and Preservation in Real Languages

- Shocking News: Many type systems obey neither progress or preservation!
- Example: C, C++
- More Shocking News: Very few type systems obey progress!
- Example: Java
- But progress is a very useful property, even if it can often only be argued for some classes of run-time errors

Conclusion

- We saw how to give typing rules
- We proved progress and preservation of a type system
- Next time: Polymorphism