Lecture Notes: Soundness

17-363/17-663: Programming Language Pragmatics (Fall 2022)

Jonathan Aldrich
jonathan.aldrich@cs.cmu.edu

> Well typed programs cannot go wrong.
>
> *Robin Milner*

In prior lectures, we have formalized both operational semantics and typing rules for a simple language. The benefit of formal definitions of semantics is that it is clear exactly how a program should be typechecked and what it should do when executed. However, it's natural to wonder whether the semantics are *right*. Did we give operational semantics for all reasonable programs? Are the typing rules consistent with the operational semantics?

One of the roles of a type system is to ensure that certain kinds of errors don't happen. In particular, we'd like to exclude one particular kind of error: applying operations to inappropriate data. For example, the built-in $+$ operator should be applicable to two numbers, and maybe strings, but it's not reasonable to add functions together. Furthermore, if we define a search function that's supposed to take a string, we would like the type system to prevent us from accidentally passing a number to it.

Of course, type systems don't try to find all errors. They typically do not try to identify logical errors, where code doesn't do what it's supposed to; that is the domain of verification tools. They also don't try to identify run-time errors such a divide by zero that are not about the kind of data being operated on. Instead, they are checking that the program is basically well-structured: all operations are passed appropriate kinds of data.

What does it mean to say that an error doesn't happen? In a real implementation, an error happens when a run-time check fails, for example producing a null pointer exception in Java. Alternatively, in languages such as C with few run-time checks, execution may be undefined by the standard, meaning that the program may output garbage data. We are modeling the operational semantics of programs mathematically, and the rules define the execution only of well-formed programs. In this kind of model, a run-time error occurs when there is no rule to apply: we say that the execution of the program is *stuck*.

Since we have a formal semantics, we ought to be able to prove things about it, and a natural thing to prove is that programs do not get stuck; they run until they produce a result value. However, we can't prove this of all programs; a moment's thought yields examples like $(x : \text{nat} \Rightarrow x)+1$ that are already stuck. We can instead prove that *well-typed* programs, i.e. programs that typecheck, do not get stuck. This is known as type soundness. Milner summarized this in his famous quote above, "Well typed programs cannot go wrong,", using "go wrong" to mean programs that are stuck.

How can we formulate such a theorem? It's difficult to do so directly, because in a small-step operational semantics, programs execute as a sequence of instructions that may be infinite.

Instead, we can prove two theorems about the way programs operate that together imply what we want. The first theorem, known as *progress*, states that if a program is well-typed, then either it is a value already, or it can take a step. That takes care of soundness for a single step of execution, but how do we reason about programs that operate for many steps? Here the second theorem is handy: called *preservation*, it states that if a program is well-typed and takes a step, then the new program is also well-typed. Applying these two theorems inductively gives us what we mean by soundness: if a program is well-typed, it will keep taking steps (by progress) and it will remain well-typed (by preservation, ensuring we can keep applying the progress theorem). This can go on forever in programs that do not terminate, or it continues until the program is reduced to a value.

Let's state and prove such theorems formally. We'll start with a progress theorem about the language we've been formalizing:

Theorem 1 (Progress)**.** *If* $\bullet \vdash e : \tau$ *then either* e *value or* $e \to e'$*.*

Proof. By induction on the derivation of $\bullet \vdash e : \tau$, with a case analysis on the last rule used.

Case $\dfrac{}{\bullet \vdash n : \text{nat}}\ \textit{T-num}$: Then we have n value by rule *value-number*.

Case *T-var*: does not apply since the environment $\Gamma = \bullet$ is empty and so a variable cannot be typechecked. (This may seem mysterious–after all most source-level programs have variables—but it is explained by the fact that in a well-typed program, variables are always substituted with values before execution reaches that program location).

Case $\dfrac{\bullet \vdash e_1 : \text{nat} \quad \bullet \vdash e_2 : \text{nat}}{\bullet \vdash e_1 + e_2 : \text{nat}}\ \textit{T-plus}$:

By the induction hypothesis we know that either e_1 value or $e_1 \to e_1'$. In the latter case, we have $e_1 + e_2 \to e_1' + e_2$ by rule *congruence-plus-left*.

In the case where e_1 value, we can apply the induction hypothesis to show that either e_2 value or $e_2 \to e_2'$ Again, in the latter case we have $e_1 + e_2 \to e_1 + e_2'$ by rule *congruence-plus-right*.

Finally, if both e_1 value and e_2 value where $e_1 = n_1, e_2 = n_2$, we have $n_1 + n_2 \to n_3$ by rule *step-plus*.

Case $\dfrac{\bullet, x : \tau_2 \vdash e_1 : \tau_1}{\bullet \vdash x : \tau_2 \Rightarrow e_1 : \tau_2 \to \tau_1}\ \textit{T-fn}$: Then we have $x : \tau_2 \Rightarrow e_1$ value by rule *value-function*.

Case *T-apply*: similar to the case for *T-plus*.

Exercise 1. Complete the case for *T-let*.

This concludes the case analysis. □

Now we will prove the second component of type soundness: that typing is preserved as programs execute.

Theorem 2 (Type Preservation)**.** *If* $\Gamma \vdash e : \tau$ *and* $e \to e'$ *then* $\Gamma \vdash e' : \tau$*.*

Proof. By induction on the derivation of $e \to e'$, with a case analysis on the last rule used.

Case $\dfrac{e_1 \to e_1'}{e_1(e_2) \to e_1'(e_2)}\ \textit{congruence-call-fn}$: By inversion on the typing derivation, we know that $\Gamma \vdash e_1 : \tau_2 \to \tau_1$ and $\Gamma \vdash e_2 : \tau_2$.

We then use the induction hypothesis to conclude that $\Gamma \vdash e_1' : \tau_2 \to \tau_1$.

The case is completed by noting that $\Gamma \vdash e_1'(e_2) : \tau_1$ by the *T-apply* rule.

Case *congruence-call-arg* is analogous.

Case $\dfrac{e_2 \text{ value}}{(x : \tau_2 \Rightarrow e_1)(e_2) \to [e_2/x]e_1}$ *step-call*: By inversion on the typing derivation, we know that $\Gamma \vdash x : \tau_2 \Rightarrow e_1 : \tau_3 \to \tau_1$ and $\Gamma \vdash e_2 : \tau_3$.

We can apply inversion again to discover that the function body can be typed as $\Gamma, x : \tau_2 \vdash e_1 : \tau_1$, where $\tau_2 = \tau_3$.

We need to prove that $\Gamma \vdash [e_2/x]e_1 : \tau_1$. But we don't have a rule that gives us this directly, and showing it will require inductive reasoning because the substitution operation $[e_2/x]e_1$ recursively goes inside e_1. Fortunately, we can prove a lemma (below) called substitution that gives us exactly the result we need based on the typing facts we got from inversion.

Case *congruence-plus-left* and *congruence-plus-right*: analogous to *congruence-call-fn*.

Case *step-plus*: follows from inversion on the typing rule *T-plus* followed by application of *T-num*.

Exercise 2. Complete the cases for *congruence-let* and *step-let*.

This concludes the case analysis. □

In order to prove preservation, we needed a lemma showing that substitution preserves typing. This relates to the hypothetical nature of our typing judgment: having $x : \tau'$ in Γ is a way of assuming that some unknown term has type τ' when we typecheck e. Substitution says that if we have some actual term e' has that type in Γ, we can use it in place of x. This can also be thought of as a modularity property of the type system: we can typecheck some term e' separately from where it is used, and we know that at run time when we plug in e', the typing properties of the system will not be broken.

Lemma 3 (Substitution)**.** *If* $\Gamma, x : \tau' \vdash e : \tau$ *and* $\Gamma \vdash e' : \tau'$ *then* $\Gamma \vdash [e'/x]e : \tau$.

Proof. By induction on the structure of e. We show three sample cases; the others are similar:

Case $e = e_1 + e_2$: By inversion of the typing judgment for e we have $\dfrac{\Gamma, x : \tau' \vdash e_1 : \text{nat} \quad \Gamma, x : \tau' \vdash e_2 : \text{nat}}{\Gamma, x : \tau' \vdash e_1 + e_2 : \text{nat}}$ *T-plus*

By the induction hypothesis we know that $\Gamma \vdash [e'/x]e_1 : \text{nat}$ and $\Gamma \vdash [e'/x]e_2 : \text{nat}$.

Then we have $\Gamma \vdash [e'/x]e_1 + [e'/x]e_2 : \text{nat}$ by the *T-plus* rule.

We can apply the definition of $[e'/x](e_1 + e_2) = [e'/x]e_1 + [e'/x]e_2$ to give us the required result that $\Gamma \vdash [e'/x](e_1 + e_2) : \text{nat}$.

Case $e = x$: In this case the variable x we are substituting is the same as the variable x in the expression. By inversion we discover that $\tau = \tau'$. We know from the statement of the theorem that $\Gamma \vdash e' : \tau'$, and e' is the result of substitution, so we have the required result.

Case $e = y$, for $y \neq x$: In this case the variable x we are substituting for is different the variable y that is the expression. So substitution has no effect.

By inversion the original expression was typechecked with the *T-var* rule, and since the substituted expression is identical and does not rely the additional variable x in the original context, it can also be typed that way.

Case $e = y : \tau_2 \Rightarrow e_1$: By inversion e was typed with the rule $\dfrac{\Gamma, x : \tau', y : \tau_2 \vdash e_1 : \tau_1}{\Gamma, x : \tau' \vdash y : \tau_2 \Rightarrow e_1 : \tau_2 \to \tau_1}$ *T-fn*

We will assume without loss of generality that the variable x we are substituting for is different from the variable y that is bound in the function expression. We can always do this because if the variables do happen to clash, we can rename the variable in the function expression, giving it another name that does not clash with x. Renaming a bound variable, and all the uses of that variable in the scope of the binding, is called α-conversion and always preserves the meaning of programs. So the substituted expression is $y : \tau_2 \Rightarrow [e'/x]e_1$.

We'd like to apply the induction hypothesis to reason about the typing of the subexpression $[e'/x]e_1$. But what we have from the premise of *T-fn* is not quite in the right form. We have $\Gamma, x : \tau', y : \tau_2 \vdash e_1 : \tau_1$ but we need $\Gamma', x : \tau' \vdash e_1 : \tau_1$ for some Γ' that might be different from Γ.

We can solve this problem with a property called *exchange*, which allows us to reorder (i.e. exchange) the variables in Γ. By applying an exchange lemma (proved below) to swap the order of x and y we can conclude that $\Gamma, y : \tau_2, x : \tau' \vdash e_1 : \tau_1$. This matches our induction hypothesis if we assume a new $\Gamma' = \Gamma, y : \tau_2$.

To apply the induction hypothesis, we need e' to be typed in this same new environment Γ'. This requires that we add the typing assumption $y : \tau_2$ to the existing context Γ in the judgment $\Gamma \vdash e' : \tau'$. We can do this with a lemma called *weakening* (again, proved below) which allows us to add new variables to a typing context in a judgment we have already proved. So weakening will give us $\Gamma, y : \tau_2 \vdash e' : \tau'$.

We can now apply the induction hypothesis to learn that $\Gamma, y : \tau_2 \vdash [e'/x]e_1 : \tau_1$. Now we can apply the *T-var* typing rule to finish the case, showing that $\Gamma \vdash [e'/x](y : \tau_2 \Rightarrow e_1) : \tau_2$

Exercise 3. Complete the case for *T-let*.

The other cases are all analogous to one of the cases given above. □

We now finish by proving the weakening and exchange lemmas used in the substitution lemma.

Lemma 4 (Weakening)**.** *If* $\Gamma \vdash e : \tau$ *and* $x \notin$ *domain*(Γ) *then* $\Gamma, x : \tau' \vdash e : \tau$*.*

Proof. by induction on the derivation of $\Gamma \vdash e : \tau$. The only interesting case is the *T-var* rule, which looks up the definition of a variable, and always chooses the last variable in the context. However, since we assume $x \notin$ *domain*(Γ), having the extra variable x in the context won't affect this lookup. The other cases follow straightforwardly from the induction hypothesis. □

Lemma 5 (Exchange)**.** *If* $\Gamma, x : \tau_x, y : \tau_y \vdash e : \tau$ *and* $y \neq x$ *then* $\Gamma, y : \tau_y, x : \tau_x \vdash e : \tau$*.*

Proof. by induction on the derivation of $\Gamma, x : \tau_x, y : \tau_y \vdash e : \tau$. The only interesting case is the *T-var* rule, which looks up the definition of a variable, and always chooses the last variable in the context. However, since we assume $y \neq x$, reordering them doesn't affect this lookup. The other cases follow straightforwardly from the induction hypothesis. □