Programming Languages: Semantics and Types

Stephanie Weirich

August 26, 2025

1 STLC

This section gives a precise definition of the syntax of the simply-typed lambda calculus, its type system and operational semantics. If you are new to programming language theory, this development provides the opportunity to introduce some of the mathematical concepts that we will be using throughout the semester, such as inductively defined grammars, recursive definitions, and proofs by structural induction.

1.1 Syntax

The syntax of the simply-typed lambda calculus is defined by a set of terms and their associated set of types. By convention, we will use e to refer to some arbitrary term and e to refer to some arbitrary type.

Definition 1.1 (Types)**.** The set of types is inductively defined by the following rules:

1. A base type, **Nat**, is a type.
2. If τ_1 and τ_2 are types, then $\tau_1 \to \tau_2$ is a type.

The type $\tau_1 \to \tau_2$ represents the type of functions that take an argument of type τ_1 and return a value of type τ_2.

Definition 1.2 (Terms)**.** The set of terms is inductively defined by the following rules:

1. A natural number k is a term.
2. A variable x is a term.
3. If e is a term and x is a variable, then $\lambda x.e$ is a term (called a lambda abstraction). The variable x is the parameter and e is the body is the body of the abstraction.
4. If e_1 and e_2 are terms, then $e_1\ e_2$ is a term (called a function application).

The definition of terms refers to two other sets: natural numbers and variables. The set of natural numbers, $\mathbb{N}$, are an infinite set of numbers 0, 1, ...; we will use i, j and k to refer to arbitrary natural numbers. We will treat variables more abstractly. We assume that there is some infinite set of variable names, called $\mathcal{V}$, and that given any finite set of variables, we can always find some variable that is not in contained in that set. (We call this variable *fresh* because we haven't used it yet.)

Now, the above is a wordy way of describing an inductively-defined grammar of abstract syntax trees. In the future, we will use a more concise notation, called Bakus-Naur form. For example, in BNF form, we can provide a concise definition of the grammars for types and terms as follows.

Definition 1.3 (STLC Syntax (concise form))**.**

$$\begin{array}{llcl}
\textit{numbers} & \emptyset\, i, j, k & \in & \mathbb{N} \\
\textit{variables} & x & \in & \mathcal{V} \\
\textit{types} & \tau & ::= & \mathbf{Nat} \mid \tau_1 \to \tau_2 \\
\textit{terms} & e & ::= & k \mid x \mid \lambda x.e \mid e_1\ e_2
\end{array}$$

Free variables Because types and terms are inductively defined sets, we can reason about them using recursion and induction principles. The recursion principle means that we define recursive functions that takes terms or types as arguments and know that the functions are total, as long as we call the functions over smaller subterms.

For example, one function that we might define calculates the set of *free* variables in a term.

Definition 1.4 (Free variables)**.** We define the operation $\mathsf{fv}(e)$, which calculates the set of variables that occur *free* in some term e, by structural recursion.

$$\begin{array}{lcll}
\mathsf{fv}(k) & = & \emptyset & \textit{emptyset} \\
\mathsf{fv}(x) & = & \{x\} & \textit{a singleton set} \\
\mathsf{fv}(e_1\ e_2) & = & (\mathsf{fv} e_1) \cup (\mathsf{fv} e_2) & \textit{union of sets} \\
\mathsf{fv}(\lambda x.e) & = & (\mathsf{fv} e) - \{x\} & \textit{remove bound variable}
\end{array}$$

Each of the lines above describes the behavior of this function on the different sorts of terms. If the argument is a natural number constant k, then it contains no free variables, so the result of the function is the $\emptyset$. Otherwise, if the argument is a single variable, then the function returns a singleton set. If the argument is an application, then we use recursion to find the free variables of each subterm and then combine these sets using an "union" operation. Finally, in the last line of this function, we find the free variables of the body of an abstraction, but then remove the argument x from that set because it does not appear free in entire abstraction.

Variables that appear in terms that are not free are called *bound.* For example, in the term $\lambda x.x\ y$, we have x bound and y free. Furthermore, some variables may occur in both bound and free positions in terms; such as x in the term $(\lambda x.x\ y)\ x$.

Renaming Here is another example of a recursively defined function. Sometimes we would like to change the names of free variables in terms.

A *renaming*, ξ, is a mapping from variables to variables. A renaming has a *domain*, $\mathsf{dom}\,\xi$ and a *range* $\mathsf{rng}\,\xi$. We use the notation y/x for a single renaming that maps x to y, and the notation $y/x, \xi$ to extend an existing renaming with a new replacement for x.

Definition 1.5 (Renaming application)**.** We define the application of a renaming to a term, written with postfix notation $e\langle\xi\rangle$, as follows:

$$\begin{array}{lcl}
k\langle\xi\rangle & = & k \\
x\langle\xi\rangle & = & \xi\, x \\
(e_1\ e_2)\langle\xi\rangle & = & (e_1\langle\xi\rangle)\ (e_2\langle\xi\rangle) \\
(\lambda x.e)\langle\xi\rangle & = & \lambda x.(e\langle x/x, \xi\rangle)
\end{array}$$

We can only apply a renaming to a term when its domain includes the free variables defined in the term. In that case, our renaming function is total: it produces an answer for any such term.

We have to be a bit careful in the last line of this definition. What if ξ already maps the variable x to some other variable? Our goal is to only rename free variables: the function should leave the bound variables alone and inside the body of $\lambda x.e$, the variable x is occurs bound, not free. By updating the renaming to $x/x, \xi$ in the recursive call, we force the renaming that we use for the body of the abstraction to not change x.

Substitution There is one final definition of a function defined by structural recursion over terms: the application of a *substitution* that applies to all free variables in the term.

A *substitution*, σ is a mapping from variables to terms. As above, it has a *domain* (a set of variables) and a *range* (this time a set of terms). We use the notation $e/x, \sigma$ to refer to the substitution that maps variable x to term e, but otherwise acts like σ.

As before, this definition only applies when the free variables of the term are contained within the domain of the substitution. Furthermore, when substituting in the body of an abstraction, we must extend the substitution with a definition for the bound variable (mapping it to itself).

Definition 1.6 (Substitution application)**.** We define the application of a substitution function to a term, written with postfix notation $e[\sigma]$, as follows:

$$\begin{array}{lcl}
k[\sigma] & = & k \\
x[\sigma] & = & \sigma\, x \\
(e_1\ e_2)[\sigma] & = & (e_1[\sigma])\ (e_2[\sigma]) \\
(\lambda x.e)[\sigma] & = & \lambda x.(e[x/x, \sigma])
\end{array}$$

Variable binding, alpha-equivalence and all that At this point, we will start to be somewhat informal when it comes to bound variables in terms. We don't really want to distinguish between terms that differ only in their use of bound variables, such as $\lambda x.x$ and $\lambda y.y$. The relation α-equivalence relates such terms, and from this point forward we will "work up-to-α-equivalence. What this means practically is that on one hand, we must be sure that our definitions don't really depend on the names of bound variables. In return, we can always assume that any bound variable is distinct from any other variable, if we need it to be.

Here we have a conundrum. It is common practice to describe lambda calculus terms as we have done above (sometimes called using a named or nominal representation of variables). But getting the details right is difficult (it requires maintaining careful invariants about all definitions) and subtle. If you are working with a proof assistant, you really do need to get the details right, but it also makes sense to use an approach (such as de Bruijn indices) where the details are easier to get right. This is what we will do in the accompanying mechanized proofs.

However, because using a named representation is standard practice we will continue to use that approach in these notes, glossing over some details. This will allow us to stay roughly equivalent to the proof scripts (which have other details). Because of the informal nature of our discussion, there will probably be minor errors related to variable naming; but we won't stress about them.

1.2 Type system

Next we will define a typing relation for STLC. This relation has the form $\Gamma \vdash e : \tau$, which is read as "in the typing context Γ, the term e has type τ." The typing context Γ, tells us what the types of free variables should be. Therefore, we can view it as a finite map from variables to types, and write it by listing all of the associations $x{:}\tau$. If a term is in this relation we say that it "type checks".

We define the typing relation inductively, using the following rules. A term type checks if we can find some tree that puts these rules together in a *derivation*. In each rule, the part below the line is the conclusion of the rule, and the rule may have multiple premises. In a derivation tree, each premise must be satisfied by subderivations, bottoming out with rules such as rule T-VAR or rule T-LIT that do not have any premises for the same relation.

Definition 1.7 (STLC type system)**.**

$\boxed{\Gamma \vdash e : \tau}$ *(in context Γ, term e has type τ)*

$$\frac{}{\Gamma \vdash k : \mathbf{Nat}}\ \textsc{t-lit} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{t-var} \qquad \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}\ \textsc{t-abs} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \textsc{t-app}$$

In the variable rule, we look up the type of the variable in the typing context. This variable must have a definition in Γ for this rule to be used. If there is no type associated with x, then we say that the variable is unbound and that the term fails to *scope-check*.

In rule T-ABS, the rule for abstractions, we type check the body of the function with a context that has been extended with a type for the bound variable. The type of an abstraction is a function type $\tau_1 \to \tau_2$, that states the required type of the parameter τ_1 and the result type of the body τ_2.

Rule T-APP, which checks the application of functions, requires that the argument to the function has the same type required by the function.

1.3 Operational Semantics

Is this type system meaningful? Our type system makes a distinction between terms that type check (such as $(\lambda x.x)\ 3$) and terms that do not, such as $(2\ 5)$. But how do we know that this distinction is useful? Do we have the right rules?

The key property that we want is called *type safety.* If a term type checks, we should be able to evaluate it without triggering a certain class of errors.

One way to describe the evaluation of programs is through a *small-step* operational semantics. This is a mathematical definition of a relation between a program e and its value. We build up a small step semantics in two parts. First, we define a single step relation, written $e \leadsto e'$, to mean that a term reduces to e in one step. Then we iterate this relation, called the multistep relation and written $e \leadsto^* e'$, to talk about all of the different programs that e could reduce to after any number of steps, including 0.

The multistep evaluations that we are interested in are the ones where we do some number of small steps and get to an e' that has a very specific form, a *value.* If we have $e \leadsto^* v$ then we say that *e evaluates to v.*

Definition 1.8 (Value)**.** A *value* is an expression that is either a natural number constant or an abstraction.

$$v ::= k \mid \lambda x.e$$

We define the single step relation inductively, using the inference rules below that state when one term steps to another.

Definition 1.9 (Small-step relation)**.**

$\boxed{e \leadsto e'}$ *(term e steps to e′)*

$$\textsc{s-beta}\ \frac{}{(\lambda x.e)\ v \leadsto e[v/x]} \qquad \textsc{s-app-cong}1\ \frac{e_1 \leadsto e_1'}{e_1\ e_2 \leadsto e_1'\ e_2} \qquad \textsc{s-app-cong}2\ \frac{e_2 \leadsto e_2'}{v\ e_2 \leadsto v\ e_2}$$

In each of these three rules, the part below the line says when the left term steps to the right term. Rule STEP-BETA describe what happens when an abstraction is applied to an argument. In this case, we substitute the argument for the parameter in the body of the function. Note in this rule that the argument must be a value before substitution. If it is not a value, then we cannot use this rule to take a step. This rule is the key of a *call-by-value* semantics.

The second two rules each have premises that must be satisfied before they can be used. Rule STEP-APP-CONGONE applies when the function part of an application is not (yet) an abstraction. Similarly, the last rule applies when the argument part of an application is not (yet) a value.

This small step relation is intended to be deterministic. Any term steps to at most one new term.

Lemma 1.1 (Determinism)**.** If $e \leadsto e_1$ and $e \leadsto e_2$ then $e_1 = e_2$.

The small step relation is *not* a function. For some terms e, there is no term e' such that $e \leadsto e'$. For example, if we have a number in the function position, e.g. $(3\ e)$, then the term does not step and these terms do not evaluate to any value.

This is important. These terms are called *stuck* and correspond to crashing programs. For example, if we tried to use a number as function pointer in the C language, then we might get a segmentation fault.

1.4 Type Safety

Type safety is a crucial property of a typed programming language. It ensures that a well-typed program will never "go wrong" during execution. For the simply-typed lambda calculus, this means a program will not get stuck in a state where it cannot take a reduction step but is not a final value.

The type safety proof is usually defined through two lemmas: Preservation and Progress.

Preservation The *preservation* lemma property states that if a term e has type τ, and it takes a single reduction step to e', then the new term must also have the exact same type τ. In other words, the type is "preserved" through evaluation.

Lemma 1.2 (Preservation)**.** If $\emptyset \vdash e : \tau$ and $e \rightsquigarrow e'$ then $\emptyset \vdash e' : \tau$.

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

- In the case of rule S-BETA, we have that e is of the form $(\lambda x.e)\ v$ and e' is $e[v/x]$. We also know that the first term type checks, i.e. $\emptyset \vdash (\lambda x.e)\ v : \tau$. For this term to type check, we must have used rule T-APP, so we also know that $\emptyset \vdash (\lambda x.e) : \tau_1 \to \tau$ and $\emptyset \vdash v : \tau_1$. (This logical step is referred to as *inversion* as we are reading a typing rule from bottom to top.). We can do this again, because the only way to make an abstraction to type check is rule rule T-ABS, so we must have also shown $x{:}\tau_1 \vdash e : \tau$. At this point we, we will appeal to a *substitution lemma* (see 1.1 below) to finish this case of the proof.
- In the case of rule S-APP-CONGONE, we have the conclusion $e_1\ e_2 \rightsquigarrow e_1'\ e_2$, and premise $e_1 \rightsquigarrow e_1'$. For the first term to type check, we again must have also used rule T-APP, so we know that $\emptyset \vdash e_1 : \tau_1 \to \tau$ and $\emptyset \vdash e_2 : \tau_1$. In this case we can use induction, because we know that e_1, a term in the subderivation both steps and type checks. So we know that $\emptyset \vdash e_1' : \tau_1$. Now we can use rule T-APP to conclude that $\emptyset \vdash e_1'\ e_2 : \tau$.
- This case is similar to the one above.

In the rule S-BETA case, our proof above relies on this lemma, that we can write more formally:

Corollary 1.1 (Single Substitution)**.** If $x{:}\tau_1 \vdash e : \tau_2$ and $\emptyset \vdash v : \tau_2$ then $\emptyset \vdash e[v/x] : \tau_1$

However, to prove this lemma, we must first generalize it. We cannot prove the lemma directly as stated, because we need a version that gives us a stronger induction hypothesis; one that works for any substitution (not just a singleton one) and any context of terms (not just one with a single variable assumption).

Lemma 1.3 ((Full) Substitution)**.** If $\Gamma \vdash e : \tau$ and for all $x \in \mathsf{dom}\,\sigma$, we have $\Delta \vdash \sigma\,x : \Gamma\,x$, then $\Delta \vdash e[\sigma] : \tau$.

Proof Sketch: The proof is by induction on the typing derivation.

Progress The second lemma, called *progress* states that any well-typed term that has not been completely reduced can always take at least one more reduction step. It ensures that a well-typed term is not "stuck." (i.e. is not a value but cannot step).

Lemma 1.4 (Progress)**.** If $\emptyset \vdash e : \tau$ then either e is a value or there exists an e' such that $e \rightsquigarrow e'$.

1.5 Is this all?

We claimed that type safety means that well-typed programs either run forever, or terminate with a value. But is that what we have really proven? Not really.

To tell the full story, we need to talk about programs that run forever. But how do we do that? So far, we only have the inductive definition of $e \rightsquigarrow^* v$, which captures finite evaluate sequences.

Now consider the following *coinductive* definition.

Definition 1.10 (Runs safely)**.** A program e *runs safely,* if it is a value or if $e \rightsquigarrow e'$, and e' *runs safely.*

This is exactly the definition we want to use in a type safety theorem.

Theorem 1.1 (Type Safety)**.** If $\emptyset \vdash e : \tau$ then e *runs safely.*

Just as in an inductive definitions, the definition of "runs safely" refers to itself. But we are interpreting this definition coinductively, so it includes both finite an infinite runs. In other words, if a program steps to another program, which steps to another program, and so on, infinitely, then it is included in this relation.

Coinductive definitions come with *coinduction* principles. We usually use induction principles to show that some property holds about an element of an inductive definition that we already have. As we "consume" this definition, we can assume, by induction, that the property is true for the subterms of the definition. For example, when proving the preservation lemma, we assumed that the lemma held for the subterms of the evaluation derivation.

The principle of coinduction applies when we want to "generate" an element of a coinductive definition. Watch!

We will prove type safety through coinduction. Given a well typed term $\emptyset \vdash e : \tau$, the progress lemma tells us that it is either a value or that it steps. If it is a value, then we know directly that it runs safely. If it steps, i.e. if we have $e \rightsquigarrow e'$, then by preservation, we know that $\emptyset \vdash e' : \tau$. By the principle of coinduction, we know that e' runs safely. So we can conclude that e runs safely.

When are we allowed to use a coinductive hypothesis? With induction, we were limited to "consuming" subterms or smaller derivations. But when we use a coinductive hypothesis, it cannot be the last step of the proof. We need to do something with the result of this hypothesis to generate our coinductive definition.

This can be a bit confusing at first, and I encourage you to look at proofs completed with coinduction in the first place to get the hang of using this principle.

An inductive definition Alternatively, if you are still uncomfortable with coinduction, we can define what it means to run safely another way.

We say that an expression e steps to e' in k steps using the following inductive definition.

Definition 1.11. $\boxed{e \rightsquigarrow^k e'}$ *(k steps)*

$$\frac{}{e \rightsquigarrow^0 e}\ \text{MS-K-REFL} \qquad \frac{e_0 \rightsquigarrow e_1 \qquad e_1 \rightsquigarrow^k e_2}{e_0 \rightsquigarrow^{1+k} e_2}\ \text{MS-K-STEP}$$

Definition 1.12 (Safe for k)**.** An expression evaluates safely for k steps if it either there is some e', such that $e \rightsquigarrow^k e'$, or there is some number of steps j strictly less than k where the term terminates with a value (i.e. there is some v and $j < k$ such that $e \rightsquigarrow^j v$).

We can now state type safety this way. We can't really talk about an infinite computation, but we can know that for an arbitrarily long time, e will run safely during that time.

Theorem 1.2. If $\emptyset \vdash e : \tau$ then for all natural numbers k, e is safe for k.

We show this result by induction on k. If k is 0, then the result is trivial. All expressions run safely for zero steps. If k is nonzero, then progress states that e is either a value or steps. If it is a value, we are also done, as values are safe for any k. If it steps to some e', then preservation tells us that $\emptyset \vdash e' : \tau$. By induction, we know that e' is safe for $k - 1$. So either $e' \rightsquigarrow^j v$, i.e. e' steps to some value v within j steps, for some $j < k - 1$, or $e' \rightsquigarrow^{k-1} e''$. In the first case, we have $e \rightsquigarrow^{j+1} v$ which is a safe evaluation for e. In the second case, we have $e \rightsquigarrow^k e''$, which is also a safe evaluation for e.