Programming Languages: Semantics and Types

Stephanie Weirich

September 9, 2025

Contents

1

Type Safety for a Simply-Typed Lambda Calculus

This section gives a precise definition of the syntax of a simply-typed lambda calculus, its type system and small-step operational semantics. For conciseness, we often refer to this language as STLC.

If you are new to programming language theory, this section also introduces 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.

STLC is actually a family of simple languages, with some freedom in the sorts of features that are included. There must always be some sort of "primitive" type such as booleans, numbers, or even a unit type. And STLC always includes first-class functions, i.e. λ-terms, making it a simplified version of typed functional languages such as ML or Haskell. However, in other contexts you may see it extended with various other features, such as records, products, disjoint unions, variant types, etc.

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 the metavariable e to refer to some arbitrary term and τ to refer to some arbitrary type. If you are familiar with algebraic datatypes, or inductive datatypes, you can think of the following definitions along those lines.

Definition 1.1.1 (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 1.1.2 (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, $\ldots$; we will use i, j and k to refer to arbitrary natural numbers. We 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 such a variable *fresh* because we haven't used it yet.) If you like, you can think of names more concretely as strings or numbers, but we won't allow all of the usual operations on strings and numbers to be applied to names.

Now, the above definitions are 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 1.1.3 (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 1.1.4 (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.

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

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

$$\begin{array}{lcl}
k\langle\xi\rangle & = & k \\
x\langle\xi\rangle & = & \xi\, x \\
(e_1\ e_2)\langle\xi\rangle & = & (e_1\langle\xi\rangle)\ (e_2\langle\xi\rangle) \\
(\lambda x.e)\langle\xi\rangle & = & \lambda y.(e\langle y/x, \xi\rangle) \text{ for } y \text{ not in } \mathsf{rng}\,\xi
\end{array}$$

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? What if ξ already maps some other variable to x? Our goal is to only rename free variables: the function should leave the bound variables alone. Inside the body of $\lambda x.e$, the variable x occurs bound, not free. On the other hand, if we introduce a new x through renaming an existing free variable, we do not want it to be *captured* by the function. For example, if we rename x to y, in the function $\lambda x.y\langle x/y\rangle$, we do not want to produce $\lambda x.x$.

Therefore, we pick some fresh variable y, and updating the renaming to $(y/x, \xi)$ in the recursive call. (If x is already fresh, we can keep using it.) That way, we force the renaming that we use for the body of the abstraction to not change the bounding structure of the term.

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 be careful to avoid variable capture.

Definition 1.1.6 (Substitution application)**.** We define the application of a substitution function to a term, written with postfix notation $e[\sigma]$, as follows:

$$
\begin{array}{lcl}
k[\sigma] & = & k \\
x[\sigma] & = & \sigma\, x \\
(e_1\ e_2)[\sigma] & = & (e_1[\sigma])\ (e_2[\sigma]) \\
(\lambda x.e)[\sigma] & = & \lambda y.(e[y/x, \sigma]) \text{ when } y \notin \mathsf{fv}(\ \mathsf{rng}\, \sigma)
\end{array}
$$

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. As you see above, we need to be careful about variable capture when doing renaming and substitution. But we don't want to pollute our reasoning later with these details.

Fortunately, we also don't want to distinguish between terms that differ only in their use of bound variables, such as $\lambda x.x$ and $\lambda y.y$. There is a relation called α-equivalence that relates such terms, and from this point forward we will say that our definitions are "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. This practice is called the "Barendregt Variable Convention"[Bar84].

But, note that this is an informal convention, allowing us to follow the common practice of describing 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. In that context, it also makes sense to use an approach (such as de Bruijn indices [de 72]) 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 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 be minor omissions 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 $\Gamma \vdash e \in \tau$, 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 1.2.1 (STLC type system)**.**

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

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

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 1.3.1 (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 1.3.2 (Small-step relation)**.**

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

$$\frac{}{(\lambda x.e)\ v \leadsto e[v/x]}\;\textsc{s-beta} \qquad \frac{e_2 \leadsto e_2'}{v\ e_2 \leadsto v\ e_2}\;\textsc{s-app-cong2} \qquad \frac{e_2 \leadsto e_2'}{v\ e_2 \leadsto v\ e_2}\;\textsc{s-app-cong2}$$

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 1.3.1 (Determinism)**.** If $e \rightsquigarrow e_1$ and $e \rightsquigarrow e_2$ then $e_1 = e_2$.

The small step relation is *not* a function. For some terms e, there is no term e' such that $e \rightsquigarrow e'$. 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 Preservation and Progress

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

We can prove this lemma in three separate ways: by structural induction on the syntax of e, or by induction on the derivations of $\emptyset \vdash e \in \tau$ or $e \rightsquigarrow e'$. This flexibility is enabled by the simplicity of this type system. For example, we have exacltly one typing rule for each syntactic form, and each typing rule has a corresponding premise for each subterm.

Because this is our first inductive proof, we will first prove it by induction on the syntax and then by induction on the step relation.

Proof. Proof is by induction on the syntax of e. We want to prove that the statement of the lemma holds for any arbitrary e. Our induction principle requires that we prove that it holds for the four different syntactic forms, natural numbers, variables, applications and abstractions. In the latter two cases, we will be able to assume that the lemma holds for each of the subterms of the form.

- If e is a natural number k, then we want to show that if $\emptyset \vdash k \in \tau$ and $k \rightsquigarrow e'$ then $\emptyset \vdash e' \in \tau$. However, by looking at our operational semantics, we see that constants don't step, so there cannot be any such e'. So this case is immediate.
- If e is a variable x, then we want to show that if $\emptyset \vdash x \in \tau$ and $x \rightsquigarrow e'$ then $\emptyset \vdash e' \in \tau$. Again, this case is impossible, and for two reasons. We cannot type check variables in an empty context and there is no rule of the operational semantics that applies.
- If e is $\lambda x.e$, then this case also impossible as abstractions don't step.
- If e is of the form $e_1\ e_2$, then we want to show that if $\emptyset \vdash e_1\ e_2 \in \tau$ and $e_1\ e_2 \rightsquigarrow e'$ then $\emptyset \vdash e' \in \tau$. In this case, we can look at the typing rules to observe that there is only one way to type check an application. For this application to type check, the derivation must also show that $\emptyset \vdash e_1 \in \tau_2 \rightarrow \tau$ and $\emptyset \vdash e_2 \in \tau_2$. (This reasoning principle is called *inversion*.) We can also use inversion on the step relation. But this time there are three ways that an application could step.
  - The application could step using rule S-BETA. In this case, we know that e_1 is $\lambda x.e_1'$ and e_2 is some value v. Furthermore, we have that the result of the application step e' is $e_1'[v/x]$. Therefore, we need to show that the result of this substitution has type τ. By inverting the typing judgement for $\emptyset \vdash \lambda x.e_1' \in \tau_2 \rightarrow \tau$, we know that $x : \tau_2 \vdash e' \in \tau$. At this point we, we will appeal to a *substitution lemma* (see 1.4.1 below) to finish this case of the proof.
  - The application could step using rule S-APP-CONGONE. In this case, we know that $e_1 \rightsquigarrow e_1'$ and that e' is $e_1'\ e_2$. Because e_1 is a subterm of $e_1\ e_2$, we can assume that the preservation holds for that term. Therefore, we know that $\emptyset \vdash e_1' \in \tau_2 \rightarrow \tau$. We then use this fact with rule T-APP to conclude that $\emptyset \vdash e_1'\ e_2 \in \tau$.
  - The application could step using rule S-APP-CONGTWO. In this case, we know that e_1 is a value and that $e_2 \rightsquigarrow e_2'$. As above, we can assume that the preservation holds for e_2. Therefore, we know that $\emptyset \vdash e_2' \in \tau_2$. We then use this fact with rule T-APP to conclude that $\emptyset \vdash e_1\ e_2' \in \tau$.

□

For comparison, we also prove this theorem by induction on one of the derivations. The proof is not significantly different: the same pieces of the argument are necessary. However, we put them together in a different way, replacing some uses of inversion with the induction principle.

Proof. The proof is by induction on the derivation of $e \rightsquigarrow e'$. 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. that $\emptyset \vdash (\lambda x.e)\ v \in \tau$. For this term to type check, we must have used rule T-APP, so by inversion, we also know that $\emptyset \vdash (\lambda x.e) \in \tau_1 \rightarrow \tau$ and $\emptyset \vdash v \in \tau_1$. 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 $x : \tau_1 \vdash e \in \tau$. At this point we, we will again appeal to the substitution lemma (see 1.4.1 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 $\emptyset \vdash e_1 \in \tau_1 \rightarrow \tau$ and $\emptyset \vdash e_2 \in \tau_1$. 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 $\emptyset \vdash e_1' \in \tau_1$. Now we can use rule T-APP to conclude that $\emptyset \vdash e_1'\ e_2 \in \tau$.
- This case is similar to the one above.

□

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

Corollary 1.4.1 (Single Substitution)**.** If $x : \tau_1 \vdash e \in \tau_2$ and $\emptyset \vdash v \in \tau_2$ then $\emptyset \vdash e[v/x] \in \tau_1$

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. To see where we run into trouble, let's walk through a potential proof and see where we get stuck.

We can try to prove this lemma by structural induction on e. That means that we have four cases. The cases for variables, constants and applications go through without difficulty. Now consider the case for abstractions. Say we have $x : \tau_1 \vdash \lambda y.e_1 \in \tau_2$. We want to show that $\emptyset \vdash (\lambda y.e_1)[v/x] \in \tau_2$. From the definition of substitution, we know that this is equivalent to showing that $\emptyset \vdash \lambda y.e_1[v/x] \in \tau_2$, implicitly using the variable convention to assume that y is not the same as x. By inversion on the typing judgement, we also know that τ_2 is some function type $\tau_3 \rightarrow \tau_4$ and that $x : \tau_1, y : \tau_3 \vdash e_1 \in \tau_4$. From rule T-ABS, it suffices to show $y : \tau_1 \vdash e_1[v/x] \in \tau_4$. However, this result is not available to us through induction: the lemma only applies to terms that type check in a context with exactly one variable. However, although e_1 is a subterm, it type checks in a context with two variables. So we cannot make any more progress on this proof.

Therefore, we generalize the substitution lemma in two ways. First, we allow the term to type check in any context Γ. Then, the lemma works for any substitution σ that replaces every variable in $\mathsf{dom}\,\Gamma$, i.e. in scope, to a term of the appropriate type. However, the range of σ need not be closed: we use Δ to describe the types of variables that can appear in the range of σ.

Lemma 1.4.2 (Simultaneous substitution)**.** If $\Gamma \vdash e \in \tau$ and for all $x \in \mathsf{dom}\,\Gamma$, we have $\Delta \vdash \sigma\, x \in \Gamma\, x$, then $\Delta \vdash e[\sigma] \in \tau$.

Proof. Proof is by structural induction on e. That means that we have four cases. Again, the cases for variables, constants and applications go through without difficulty. Now consider the case for abstractions. Say we have $\Gamma \vdash \lambda y.e_1 \in \tau_2$. We want to show that $\emptyset \vdash (\lambda y.e_1)[\sigma] \in \tau_2$.

From the definition of substitution, we know that this is equivalent to showing that $\emptyset \vdash \lambda y.(e_1[y/y, \sigma]) \in \tau_2$, implicitly using the variable convention to assume that y is not in the domain of σ or free in the range of σ. (The substitution $(y/y, \sigma)$ is a map that is just like σ, but maps the variable y to itself.) By inversion on the typing judgement, we also know that τ_2 is some function type $\tau_3 \to \tau_4$ and that $\Gamma, y : \tau_3 \vdash e_1 \in \tau_4$. From rule T-ABS, it suffices to show $\Delta, y : \tau_3 \vdash e_1[y/y, \sigma] \in \tau_4$. This time we can use our inductive hypothesis on the typing derivation for e_1. However, to do so, we need to show that for all x in $\mathsf{dom}\,\Gamma, y : \tau_3$, we have $\Delta, y : \tau_3 \vdash (y/y, \sigma)\, x \in (\Gamma, y : \tau_3)\, x$. But we know that $\Delta, y : \tau_3 \vdash y \in \tau_3$ and we already know that the rest of the substitution is well-typed. □

This is not the only way to strengthen our substitution lemma. If our typing contexts are not ordered we can stick with single substitutions. Not ordered means that we consider $\Gamma, x : \tau_1$ to be the same context as $x : \tau_1, \Gamma$.

Lemma 1.4.3 (Substitution (Unordered context))**.** *If $\Gamma, x : \tau_1 \vdash e \in \tau$ and $\Gamma \vdash v \in \tau_1$, then $\Gamma \vdash e[v/x] \in \tau$.*

If contexts are ordered, then another way to strengthen this lemma is to let the variable being substituted for appear anywhere in the middle of the context. However, the proof of this version of the lemma requires an additional property called weakening (shown below).

Lemma 1.4.4 (Substitution (Ordered context))**.** *If $\Gamma, x : \tau_1, \Gamma' \vdash e \in \tau$ and $\Gamma \vdash v \in \tau_1$, then $\Gamma, \Gamma' \vdash e[v/x] \in \tau$.*

Lemma 1.4.5 (Weakening)**.** *If $\Gamma \vdash e \in \tau$ then $\Gamma, \Gamma' \vdash e \in \tau_1$.*

Weakening is a corollary of our strongest substitution lemma where we set $\Delta = \Gamma, \Gamma'$ and σ to be the identity function.

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

We prove this lemma by induction in the typing derivation. In the rules where e is already a value, then the proof is trivial. Therefore we only need to consider when e is an application of the form $e_1\ e_2$, where $\emptyset \vdash e_1 \in \tau_1 \to \tau$ and $\emptyset \vdash e_2 \in \tau_1$. By induction on the first premise, we know that either e_1 is a value or that it takes a step to some e_1'. If it takes a step, the entire application takes a step by rule S-APP-CONG1 and we are done. Otherwise, if it is a value, then we know that it must be of the form $\lambda x.e'$, because it must have a function type. By induction on the second premise, we know that either e_2 is a value or that it takes a step to some e_2'. In the former case, the application steps to $e'[e_2/x]$ by rule S-BETA, in the latter case, the application steps to $(\lambda x.e')\ e_2'$ by rule S-APP-CONG2.

1.5 What is type safety?

We above claimed that type safety means that well-typed programs do not get stuck. But what does this mean? Is that what we have really proven?

There are languages and type systems that do not satisfy both of these lemmas, yet we still might like to say that they are type safe. Can we come up with a more general definition? Something that is implied by preservation/progress but doesn't itself require them to be true.

Perhaps we would like to prove something like below, where the multistep relation $\leadsto^*$ is iteration of the single-step relation any number of times. If a closed term type checks then it must evaluate to a value with the same type.

Conjecture 1.5.1 (Terminating Type Safety)**.** If $\emptyset \vdash e \in \tau$ then there exists some value v such that $e \leadsto^* v$ and $\emptyset \vdash v \in \tau$.

This conjecture seems straightforward to prove from progress and preservation. By progress we know that either a term is a value or that it steps. By preservation, we know that it if it steps, it has the same type. But what we are missing from a straightforward proof is the fact that this conjecture says that evaluation *terminates*. How do we know that we will eventually reach a value in some finite number of steps?

It turns out that this conjecture is true, but we are not yet ready to prove it directly. But even though the conjecture is true, it is not a good definition of type safety: even though all well-typed STLC programs halt, that is not true of most programming languages. And we would like to have a definition of type safety that also applies to those languages. One that shows that well-typed programs do not get stuck, while not requiring them to produce values.

There are several solutions to this issue.

Well-typed programs don't get stuck The most straightforward approach is to define what it means for a program to get stuck, and then show that this cannot happen.

Definition 1.5.1 (Stuck)**.** A term e is *stuck* if it is not a value and there does not exists any e' such that $e \leadsto e'$.

Theorem 1.5.1 (Type safety (no stuck terms))**.** If $\emptyset \vdash e \in \tau$ then for all e', such that $e \leadsto^* e'$, e' is not stuck.

Proof. We prove this by induction on the derivation of $e \leadsto^* e'$. If there are no steps in this reduction sequence, then e is equal to e'. By the progress lemma, we know that e is not stuck. Otherwise, say that there is at least one step, i.e. there is some e_1 such that $e \leadsto e_1$ and $e_1 \leadsto^* e'$. By preservation, we know that $\emptyset \vdash e_1 \in \tau$. Then we can use induction to say that e' is not stuck. □

A coinductive definition What if we want to state type safety a little more positively. In other words, we want to say that a well typed term either produces a value or runs forever, without having to talk about stuckness.

We can do that using the following *coinductive* definition.

Definition 1.5.2 (Runs safely)**.** A program e *runs safely*, if it is a value or if $e \leadsto e'$, and e' *runs safely*.

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

Theorem 1.5.2 (Type Safety (runs safely))**.** If $\emptyset \vdash e \in \tau$ 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 $\emptyset \vdash e \in \tau$, 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 $\emptyset \vdash e' \in \tau$. 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 1.5.3. $\boxed{e \rightsquigarrow^k e'}$ *(k steps)*

$$\text{MS-K-REFL}\quad \frac{}{e \rightsquigarrow^0 e} \qquad\qquad \text{MS-K-STEP}\quad \frac{e_0 \rightsquigarrow e_1 \qquad e_1 \rightsquigarrow^k e_2}{e_0 \rightsquigarrow^{\mathbf{S}\,k} e_2}$$

Definition 1.5.4 (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 using this step-counting definition. 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 1.5.3 (Type Safety (step-counting))**.** If $\emptyset \vdash e \in \tau$ 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 $\emptyset \vdash e' \in \tau$. 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.

1.6 Further reading

The type safety proof for the simply-typed lambda calculus is explained in a number of textbooks including TAPL [Pie02], PFPL [Har16] and Software Foundations [PdAC$^+$25]. Each of these sources defines type safety as the conjunction of preservation and progress.

Milner [Mil78] proved a *type soundness* theorem, which states that well-typed ML programs cannot “go wrong”. To do so, he constructed a denotational semantics of the ML language that maps every ML program to either some mathematical value (like a number or continuous function), to a special element indicating divergence ($\bot$), or to a special element called “wrong” that indicates a run-time error. He then proved that if a program type checks, then its denotation does not include the “wrong” element.

Wright and Felleisen [WF94] observed that run-time errors could be ruled out by using a small-step operational semantics. They defined syntactic type soundness as showing preservation (inspired by subject reduction from combinatory logic), characterizing “stuck” or “faulty” expressions, and then showing that faulty expressions are not typeable (i.e. progress). They put these together with a strong soundness theorem that says that well-typed programs either diverge or reduce to values of the appropriate type.

2

Natural number recursion

STLC is rather *simple*. It lacks the computational power of most typed programming languages. All STLC expressions terminate! In due time, we will extend this language with arbitrary recursive definitions, which make the language Turing complete.

However, before we do that let's extend this system with a limited form of recursion. Our definition of STLC includes the *natural numbers* as constants, i.e. numbers starting from zero. Natural numbers can be defined using an *inductive datatype*. Any natural number is either zero or the successor of some natural number.

Let's redefine the syntax of natural numbers to make this structure explicit.

$$k \in \mathbb{N} ::= 0 \mid \mathbf{S}\,k$$

Now, instead of saying 1, or 2, or 3, we could say $\mathbf{S}\,0$, or $\mathbf{S}\,(\mathbf{S}\,0)$, or $\mathbf{S}\,(\mathbf{S}\,(\mathbf{S}\,0))$. Isn't that better? Ok, perhaps maybe not. We will keep the syntax 1, 2, 3 around for clarity, but remember that these arabic numerals stand for the unary structure.

The advantage of working with the inductive structure of natural numbers is that they now come with an induction principle (for reasoning mathematically) and a recursion principle (for creating new definitions). This is the justification that we used in the previous section for the step-counting definition of type safety.

Now that we have observed the inductive structure of natural numbers, let's add a *primitive recursion* operator to STLC that can be used to define programs, in that language, by recursion. This form of recursion will always be bounded; we will be able to write more interesting computations but will still be able to know that all experssions terminate.

In fact, we will add *two* new expression forms in this section, as shown in the grammar below.

$$e ::= \ldots \mid \mathbf{succ}\,e \mid \mathbf{nrec}\,e\,\mathbf{of}\,\{0 \Rightarrow e_0;\,\mathbf{S}\,x \Rightarrow e_1\}$$

Along with these new expression forms, we also add the typing and small-step semantics rules shown in Figure 2.1.

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

T-SUCC

$$\frac{\Gamma \vdash e \in \mathbf{Nat}}{\Gamma \vdash \mathbf{succ}\, e \in \mathbf{Nat}}$$

T-NREC

$$\frac{\begin{array}{c}\Gamma \vdash e \in \mathbf{Nat} \\ \Gamma \vdash e_0 \in \tau \\ \Gamma, x : \mathbf{Nat} \vdash e_1 \in \tau \to \tau\end{array}}{\Gamma \vdash \mathbf{nrec}\, e\, \mathbf{of}\, \{0 \Rightarrow e_0;\ \mathbf{S}\, x \Rightarrow e_1\} \in \tau}$$

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

S-SUCC-LIT

$$\frac{}{\mathbf{succ}\, k \leadsto \mathbf{S}\, k}$$

S-SUCC-CONG

$$\frac{e \leadsto e'}{\mathbf{succ}\, e \leadsto \mathbf{succ}\, e'}$$

S-NREC-ZERO

$$\frac{}{\mathbf{nrec}\, 0\, \mathbf{of}\, \{0 \Rightarrow e_1;\ \mathbf{S}\, x \Rightarrow e_2\} \leadsto e_1}$$

S-NREC-SUCC

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

S-NREC-CONG

$$\frac{e \leadsto e'}{\mathbf{nrec}\, e\, \mathbf{of}\, \{0 \Rightarrow e_1;\ \mathbf{S}\, x \Rightarrow e_2\} \leadsto \mathbf{nrec}\, e'\, \mathbf{of}\, \{0 \Rightarrow e_1;\ \mathbf{S}\, x \Rightarrow e_2\}}$$

Figure 2.1: Natural number operations: successor and recursion

The successor operation **succ** e, adds one to its argument. This operation is specified by the two rules of the operational syntax that trigger when the argument is a literal value (rule S-SUCC-LIT) and when the argument itself steps (rule S-SUCC-CONG). The typing rule (rule T-SUCC) requires the argument to have type **Nat** and asserts that its successor is also a natural number. (Note: don't confuse the successor operation **succ** e of the expression language, with the syntax **S** k of natural numbers. The former is never a value (it steps by one of the two rules) while the latter is a way of writing a natural number).

The (primitive) recursion operation $\mathbf{nrec}\, e\, \mathbf{of}\, \{0 \Rightarrow e_0;\ \mathbf{S}\, x \Rightarrow e_1\}$ recurses over e. This operation compares e to see of it is 0 or some larger number. In the first case, the expression steps to e_0. If the argument is equal to **S** k for some k, then the expression steps to e_1 where k replaces x. But that is not all! The rule also applies the result of this substitution to the recursive execution of the loop on k.

For example, we can define a doubling function on natural numbers with the following definition.

$$\mathit{double}\ x = \mathbf{nrec}\, x\, \mathbf{of}\, \{0 \Rightarrow 0;\ \mathbf{S}\, y \Rightarrow \lambda z.\mathbf{succ}\, (\mathbf{succ}\, z)\}$$

Here's how this doubling function might evaluate when given the number 2:

$$
\begin{aligned}
\mathit{double}\ 2 &= \mathbf{nrec}\,2\,\mathbf{of}\,\{0 \Rightarrow 0;\ \mathbf{S}\,y \Rightarrow \lambda z.\mathbf{succ}\,(\mathbf{succ}\,z)\} \\
&\leadsto (\lambda z.\mathbf{succ}\,(\mathbf{succ}\,z)) \\
&\qquad (\mathbf{nrec}\,1\,\mathbf{of}\,\{0 \Rightarrow 0;\ \mathbf{S}\,y \Rightarrow \lambda z.(\mathbf{succ}\,(\mathbf{succ}\,z))\}) \\
&\leadsto (\lambda z.\mathbf{succ}\,(\mathbf{succ}\,z)) \\
&\qquad ((\lambda z.(\mathbf{succ}\,(\mathbf{succ}\,z)))\ (\mathbf{nrec}\,0\,\mathbf{of}\,\{0 \Rightarrow 0;\ \mathbf{S}\,y \Rightarrow \lambda z.(\mathbf{succ}\,(\mathbf{succ}\,z))\})) \\
&\leadsto (\lambda z.\mathbf{succ}\,(\mathbf{succ}\,z))\ ((\lambda z.(\mathbf{succ}\,(\mathbf{succ}\,z)))\ 0) \\
&\leadsto (\lambda z.\mathbf{succ}\,(\mathbf{succ}\,z))\ (\mathbf{succ}\,(\mathbf{succ}\,0)) \\
&\leadsto (\lambda z.\mathbf{succ}\,(\mathbf{succ}\,z))\ (\mathbf{succ}\,1) \\
&\leadsto (\lambda z.\mathbf{succ}\,(\mathbf{succ}\,z))\ 2 \\
&\leadsto \mathbf{succ}\,(\mathbf{succ}\,2) \\
&\leadsto \mathbf{succ}\,3 \\
&\leadsto 4
\end{aligned}
$$

Because the successor case is applied to the recursive execution of the loop, the typing rule requires that it have a function type.

Why do we specify the operation in this way? Sometimes you may see a semantics for primitive recursion that directly substitutes the result of the recursive execution in the successor case instead of indirectly doing so via application. The reason is that we want a *call-by-value* semantics for iteration. The rules should fully evaluate the recursive call on the predecessor before evaluating e_1. Because our operational semantics for application is already call-by-value, we get this behavior automatically.

Note that the way that we have defined this natural number recursor through pattern matching makes it particularly simple to define a *predecessor* function:

$$\mathit{pred}\ x = \mathbf{nrec}\,x\,\mathbf{of}\,\{0 \Rightarrow 0;\ \mathbf{S}\,y \Rightarrow y\}$$

This is not the case for all recursion principles. A more restricted form, sometimes called *iteration* does not bind y in the successor case.

These new extensions satisfy the properties of *substitution*, *progress*, and *preservation* that we saw in the previous chapter. As an exercise, you might try to extend those proofs with appropriate new cases.

2.1 Further Reading

This chapter is adapted from Chapter 9 of Harper [Har16], with the recursor modified to for our call-by-value semantics. Harper calls this language to Gödel's System T [G5̈8], designed to study the consistency of arithmetic. The terminology that we use for "primitive recursion" is not quite the same as the related concept of the same name from computability theory. In that context, the natural number recursor is restricted to produce functions with types of the form $\mathbf{Nat} \rightarrow \mathbf{Nat} \rightarrow \ldots \rightarrow \mathbf{Nat}$, i.e. functions that take any number of naturals as arguments and return a natural number. This operator does not have that restriction, and can define functions that are not usually considered "primitive recursive".

In general, primitive recursion is not just for natural numbers. Any inductive type, such as lists or trees, can be equipped with its own primitive recursion operation (see Mendler's dissertation [**?**]).

3

Big-Step Operational Semantics

So far, we have only considered small-step operational semantics for STLC. This semantics is useful because it provides a substitution-based, step-by-step explanation of how each expression evaluates.

But, as we have seen, our operational semantics is deterministic! Why didn't we express the semantics using an interpreter instead.

Consider the following definition in the Roq programming language. We represent the syntax of the language using the `Tm` datatype and the interpreter using the `eval` function.

```
Inductive Tm : Type :=
    var : Var -> Tm            (* variable, with names of type Var *)
  | abs : Var -> Tm -> Tm      (* abstraction *)
  | app : Tm -> Tm -> Tm       (* application *)
  | lit : nat -> Tm.           (* literal natural number constant *)

Fixpoint eval (e : Tm) : option Tm :=
  match e with
  | var x     => None
  | lit k     => Some (lit k)
  | abs x e1  => Some (abs x e1)
  | app e1 e2 =>
      match eval e1 , eval e2 with
      | Some (abs x e1') , Some v2 => eval (e1' [v2 / x])
      | _ , _ => None
      end
end.
```

The interpreter is partial because we may try to evaluate an expression with a free variable, or because we may have a type error. In either case, the interpreter returns `None`. Otherwise, when evaluating an application, the interpreter evaluates the function to some abstraction value, the argument to some other value and then calls itself recursively after substituting the argument for the parameter.

But this definition is **not** accepted by Rocq. It rejects the definition of `eval` with the following error message:

```
Recursive call to eval has principal argument equal to
   "e1'[v2/x]" instead of one of the following variables:
   "e1" "e2".
```

The reason for this error is that Rocq assumes that we are defining this evaluation function via *structural recursion* on expressions. That means that we are allowed to call `eval` on any subterm of the argument (such as `e1` and `e2`, the subterms in the application case). However, the third recursive call is not to a subterm—instead it is to the body of the closure. Rocq cannot determine that this function terminates, so it must reject this definition.

3.1 Big-step semantics

To work around this issue in our metalogic, we can work with a *relational* version of `eval` instead. We define the inductive relation $e \Rightarrow v$ that holds when e evaluates v.

Definition 3.1.1 (Big-step semantics)**.**

$\boxed{e \Rightarrow v}$ *(term e big-steps to v)*

$$\textsc{bs-val}\ \frac{}{v \Rightarrow v} \qquad \textsc{bs-app}\ \frac{\begin{array}{c} e_1 \Rightarrow \lambda x.e_1' \qquad e_2 \Rightarrow v_1 \\ e_1'[v_1/x] \Rightarrow v_2 \end{array}}{e_1\ e_2 \Rightarrow v_2}$$

Notably, this definition requires only two rules! The first rule states that values evaluate to themselves. The second evaluates an application and holds when the function evaluates to an abstraction, the argument evaluates to a value and the substitution of the argument the parameter also evaluates to a value.

Theorem 3.1.1. Equivalence of semantics] For closed expressions e, we have $e \leadsto^* v$ if and only if $e \Rightarrow v$.

We prove each direction of this lemma separately. For the forward direction (small-step implies big-step), we need to show the following lemma:

Lemma 3.1.1. If $e \leadsto e'$ then for forall v, if $e' \Rightarrow v$ then $e \Rightarrow v$.

For the backwards direction (big-step implies small-step), we need to define multi-step analogues of the two evaluation rules of the big-step semantics.

Lemma 3.1.2 (s_val)**.** $v \leadsto^* v$

Lemma 3.1.3 (s_app)**.** If $e_1 \leadsto^* \lambda x.e_1'$ and $e_2 \leadsto^* v_1$ and $e_1'[v_1/x] \leadsto^* v_2$ then $e_1\ e_2 \leadsto^* v_2$.

This lemma itself relies on showing multi-step analogues of the single-step congruence rules for the small step semantics.

Lemma 3.1.4 (ms_app_cong1)**.** If $e_1 \leadsto^* e_1'$ then $e_1\ e_2 \leadsto^* e_1'\ e_2$.

Lemma 3.1.5 (ms_app_cong2)**.** If $e_2 \leadsto^* e_2'$ then $v_1\ e_2 \leadsto^* v_1\ e_2'$.

3.2 Big-step semantics and type safety?

The big-step semantics has fewer rules, and in some situations, may be easier to understand as it is more directly connected to an interpreter.

However, there is a significant drawback of a big-step semantics: it handles both the partiality of a runtime type error and the potential partiality of divergence in exactly the same way.

But, we run into difficulty when stating and proving type safety for this semantics. While preservation holds for this semantics:

Lemma 3.2.1 (Preservation)**.** If $e \Rightarrow v$ and $\emptyset \vdash e \in \tau$ then $\emptyset \vdash v \in \tau$.

There is no way to define an analogue for the progress lemma for the big step semantics.

It is tempting to define type safety as follows:

Conjecture 3.2.1 (Big Step Safety)**.** If $\emptyset \vdash e \in \tau$ then $e \Rightarrow v$ and $\emptyset \vdash v \in \tau$.

But this is a strong lemma — it rules out both forms of partiality. We know that the program doesn't crash, but we also know that the program doesn't diverge either. While this lemma is true for STLC, proving this lemma is not a straightforward induction. And, if we were to extend the language to include nontermination, it would no longer be true.

Bibliography

[Bar84] Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Sole distributors for the U.S.A. and Canada, Elsevier Science Pub. Co., New York, N.Y., 1984.

[de 72] N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972.

[G5̈8] Kurt Gödel. Über eine bisher noch nicht benützte erweiterung des finiten standpunktes. *Dialectica*, 12(3):280, 1958.

[Har16] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, USA, 2nd edition, 2016.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[PdAC$^+$25] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. *Programming Language Foundations*, volume 2 of *Software Foundations*. Electronic textbook, 2025. Version 6.7, `http://softwarefoundations.cis.upenn.edu`.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[WF94] A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.