Programming Languages: Semantics and Types

Stephanie Weirich

September 29, 2025

Contents

Preface

This document is a work in progress and summarizes lectures given in CIS 7000-1 during the Fall 2025 semester at the University of Pennsylvania. Expect significant changes to the material over the course of this semester.

The website for the course is:

`https://sweirich.github.io/pl-semantics-and-types/`

The source files for this document are available (`https://github.com/sweirich/pl-semantics-and-types/tree/main/notes`) and have been processed using the LaTeX and Ott [SNO+07] tools. Comments, typos and suggestions are appreciated.

All mathematical definitions and proofs have been mechanized using the Rocq proof assistant, with the assistance of the `autosubst-ocaml` tool. The proof scripts are available at `https://github.com/sweirich/pl-semantics-and-types/tree/main/rocq` and the text includes hyperlinks to each corresponding part of the artifact.

Acknowledgement: These lecture notes draw inspiration from a number of sources, most notably Harper's *Practical Foundations for Programming Languages* [Har16] and Pierce's *Types and Programming Languages* [Pie02].

What is this course all about?

This semester, we are going to consider two big questions that one might ask about programs in a programming language.

1. **When are two programs equivalent?**

   This question is the heart of programming language semantics. Our goal is to understand what programs mean, so understanding when a program is equivalent to some result of evaluation is a way of defining meaning.

   However, program equivalence is more that that. It is also essential for software development, for compiler correctness, for program verification, and for security analysis.

   For example, when you refactor a program, you are replacing a part of it with "equivalent" code: you want the code to still work, but be more general in some way. Or when a compiler optimizes your program, it replaces part of the code with an "equivalent" but faster sequence of instructions. Program verification often means showing that code is "equivalent" to some mathematical specification. And security analysis is showing that code is not equivalent to programs with certain security flaws.

   What makes a good definition of equivalence? How can we reason about this definition mathematically? How can we use this definition in practice?

2. **What does it mean to type check a program?**

   Many programming languages come with static type systems, such as Rust, Java, OCaml or Rocq. Where do these type systems come from? What does it mean when a program type checks? What does it mean when we say that a language is type safe? What can can we model using static types? How do types interact with the definition of equivalence?

How will we study these two questions?

Studying these two questions is a study in definitions. And constructing definitions involves design work. A given definition may not be intrinsically wrong, it just may not be useful. So to evaluate our designs we need to identify what we want to do with these definitions and judge them using that metric.

In general, we will study these questions using the tools of programming language theory. This means that we will model idealized versions of programming languages using mathematical objects and then prove properties about these definitions, using the techniques such as induction and coinduction.

How will we stay grounded?

Programming language theory can seem both *trivial* (only applying to tiny "toy" languages) and at the same time *esoteric* (filled with unfamiliar jargon).

Programming languages found "in the wild" are complex and rapidly changing. It can be the work of several years to complete a mathematical specification of a language. A notable example is Andreas Rossberg's work on the semantics of WebAssembly, which you can find more about by watching his keynote from ICFP 23[1].

While such work is important, it does not suit our purposes. Instead we need programming language models that we can understand in hours, not years. We need these models to be representative (i.e. they should describe common features of many different programming languages) and illustrative (i.e. they should describe these features independently of other language constructs). Just as biologist gain understanding through the study of a model organisms such yeast, nematodes or fruit flies, that capture the essence of cell biology, neuroscience, and genetics, we will look at model programming languages that capture the essence of computation, using functions, data structures, control effects, and mutability.

To make sure that we can transfer what we learn from these essential models, we will talk about their connection to languages that you already know, their extension with new features, and their ability to do more than immediately apparent through macro encoding and compilation. Some properties that we prove for small languages will immediately transfer to larger contexts. Others provide a blueprint for how we might redo similar proofs at scale. And still others don't scale, so we also need to be aware of the limitations of our work and when we need to adopt new approaches.

The unfamiliar vocabulary of any discipline is a sign of maturity. It means that researchers have examined and analyzed programming languages for more than seven decades. In the process they have made discoveries, and communicating those discoveries requires precise naming and notation. One goal of this semester is to learn these concepts, along with their unfamiliar names, and understand how they may be put to use in practice. By the end of the semester, you should be better equipped to talk about languages and read research papers about programming language advances.

What is the role of proof assistants?

If you have read Software Foundations [PdAC+25] or have taken a course like CIS 5000[2], you have already used the metalanguage of the Rocq proof assistant to mathematically model small programming languages, such as the simply-typed lambda calculus.

I am a *strong* believer in the value of these tools. They both solidify your understanding of the metalogic that we are working in as well as give immediate feedback on your progress. They are also fun to use.

However, this is *not* a course on the use of proof assistants. You do not need to have experience with Rocq or any similar tool in order to benefit from this course. The homework assignments will be in LaTeX and the exams will be on paper and

[1] `https://www.youtube.com/watch?v=Lb45xIcqGjg`

[2] `https://www.seas.upenn.edu/~cis5000/current/index.html`

will cover topics related to programming language theory.

That said, I will ensure that the material that we study this semester is amenable to mechanical development. I will be developing and checking the topics that we cover throughout the semester using Rocq and will make my code available. (Caveat: I hope that I can keep up!) You can use this code as the basis for the homework assignments and I will gladly answer any questions and provide assistance during office hours. If you want to learn how to use Rocq, this is a good opportunity. I will also assist if you would like to translate this code to another framework (such as Agda or LEAN).

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))**.**

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

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.

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

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 $[\![(e/x, \sigma) \in]\!]$ 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." If a term is in this relation we say that it "type checks". The typing context Γ, tells us what the types of free variables should be. Therefore, we model it as a finite map from variables to types, and write it by listing all of the associations, for example $x\!:\!\tau_1, y\!:\!\tau_2, z\!:\!\tau_3$. If the context has no associations, we call it the *empty context* and leave it blank, writing for example $\vdash 2 \in \mathbf{Nat}$. Terms that type check with an empty context are call *closed* and have no free variables.

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)*

$$\text{T-LIT}\ \frac{}{\Gamma \vdash k \in \textbf{Nat}} \qquad \text{T-VAR}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x \in \tau} \qquad \text{T-ABS}\ \frac{\Gamma, x{:}\tau_1 \vdash e \in \tau_2}{\Gamma \vdash \lambda x.e \in \tau_1 \to \tau_2} \qquad \text{T-APP}\ \frac{\Gamma \vdash e_1 \in \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 \in \tau_1}{\Gamma \vdash e_1\ e_2 \in \tau_2}$$

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')*

$$\text{S-BETA}\ \frac{}{(\lambda x.e)\ v \leadsto e[v/x]} \qquad \text{S-APP-CONG2}\ \frac{e_2 \leadsto e_2'}{v\ e_2 \leadsto v\ e_2} \qquad \text{S-APP-CONG2}\ \frac{e_2 \leadsto e_2'}{v\ e_2 \leadsto v\ e_2}$$

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 1.3.1 (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 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 $\vdash e \in \tau$ and $e \leadsto e'$ then $\vdash e' \in \tau$.

We can prove this lemma in three separate ways: by structural induction on the syntax of e, or by induction on the derivations of $\vdash e \in \tau$ or $e \leadsto e'$. 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 $\vdash k \in \tau$ and $k \rightsquigarrow e'$ then $\vdash e' \in \tau$. 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 $\vdash x \in \tau$ and $x \rightsquigarrow e'$ then $\vdash e' \in \tau$. 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 $\vdash e_1\ \ e_2 \in \tau$ and $e_1\ \ e_2 \rightsquigarrow e'$ then $\vdash e' \in \tau$. 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 $\vdash e_1 \in \tau_2 \rightarrow \tau$ and $\vdash e_2 \in \tau_2$. (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 $\vdash \lambda x.e_1' \in \tau_2 \rightarrow \tau$, 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 $\vdash e_1' \in \tau_2 \rightarrow \tau$. We then use this fact with rule T-APP to conclude that $\vdash e_1'\ e_2 \in \tau$.
  - 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 $\vdash e_2' \in \tau_2$. We then use this fact with rule T-APP to conclude that $\vdash e_1\ e_2' \in \tau$.

□

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 $\vdash (\lambda x.e)\ v \in \tau$. For this term to type check, we must have used rule T-APP, so by inversion, we also know that $\vdash (\lambda x.e) \in \tau_1 \rightarrow \tau$ and $\vdash v \in \tau_1$. 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 $\vdash e_1 \in \tau_1 \to \tau$ and $\vdash e_2 \in \tau_1$. 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 $\vdash e_1' \in \tau_1$. Now we can use rule T-APP to conclude that $\vdash e_1'\ e_2 \in \tau$.

- 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 $\vdash v \in \tau_2$ then $\vdash e[v/x] \in \tau_1$

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 $\vdash (\lambda y.e_1)[v/x] \in \tau_2$. From the definition of substitution, we know that this is equivalent to showing that $\vdash \lambda y.e_1[v/x] \in \tau_2$, 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 $\tau_3 \to \tau_4$ 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 $\vdash (\lambda y.e_1)[\sigma] \in \tau_2$.

From the definition of substitution, we know that this is equivalent to showing that $\vdash \lambda y.(e_1[y/y,\sigma]) \in \tau_2$, 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. $\square$

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 $\Gamma, \Gamma' \vdash e[v/x] \in \tau$.

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

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

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 $\vdash e \in \tau$ then either e is a value or there exists an e' such that $e \rightsquigarrow e'$.

Proof. We prove this lemma by induction in the typing derivation. In the rules where e is already a value, then the proof is trivial. Therefore we only need to consider when e is an application of the form $e_1\ \ e_2$, where $\vdash e_1 \in \tau_1 \to \tau$ and $\vdash e_2 \in \tau_1$. 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. $\square$

Part of this proof involves inferring the structure of a closed value from its type. Although this is a straightforward bit of reasoning by inversion, this is a key step of any progress proof. Therefore, we explicitly state the lemma (and its analogue for natural numbers below).

Lemma 1.4.7 (Canonical forms (arrow types))**.** If $\vdash v \in \tau_1 \to \tau_2$ then v is some $abs\ e$.

Lemma 1.4.8 (Canonical forms (nat))**.** If $\vdash v \in \mathbf{Nat}$ then v is some natural number k.

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 $\vdash e \in \tau$ then there exists some value v such that $e \leadsto^* v$ and $\vdash v \in \tau$.

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 $\vdash e \in \tau$ 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 $\vdash e_1 \in \tau$. 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 $\vdash e \in \tau$ 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 $\vdash e \in \tau$, 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 $e \leadsto e'$, then by preservation, we know that $\vdash e' \in \tau$. 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 (Counted steps)**.** $\boxed{e \leadsto^k e'}$ *(k steps)*

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

Definition 1.5.4 (Safe for k)**.** An expression evaluates safely for k steps if it either there is some e', such that $e \leadsto^k e'$, 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 $e \leadsto^j v$).

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 $\vdash e \in \tau$ 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 $\vdash e' \in \tau$. By induction, we know that e' is safe for $k-1$. So either $e' \leadsto^j v$, i.e. e' steps to some value v within j steps, for some $j < k-1$, or $e' \leadsto^{k-1} e''$. In the first case, we have $e \leadsto^{j+1} v$ which is a safe evaluation for e. In the second case, we have $e \leadsto^k e''$, 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 ($\perp$), 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.

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

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 this unary structure.

The advantage of working with an 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 induction principle is the justification that we used in the previous section for the step-counting definition of type safety. Natural number induction is a common proof technique so we will see more of in the future!

Now that we have observed the inductive structure of natural numbers, let's incorporate this structure into our programming language so that STLC programs can work with natural numbers.

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

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

The first form, written $\mathbf{succ}\, e$, lifts the natural number successor $\mathbf{S}\, k$ to be a primitive operation. Instead of only being able to take the successor of a literal

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

$$\text{T-SUCC}\quad \frac{\Gamma \vdash e \in \mathbf{Nat}}{\Gamma \vdash \mathbf{succ}\, e \in \mathbf{Nat}}$$

$$\text{T-NREC}\quad \frac{\Gamma \vdash e \in \mathbf{Nat} \qquad \Gamma \vdash e_0 \in \tau \qquad \Gamma, x\!:\!\mathbf{Nat} \vdash e_1 \in \tau \to \tau}{\Gamma \vdash \mathbf{nrec}\, e\, \mathbf{of}\, \{0 \Rightarrow e_0;\ \mathbf{S}\, x \Rightarrow e_1\} \in \tau}$$

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

$$\text{S-SUCC-LIT}\quad \frac{}{\mathbf{succ}\, k \rightsquigarrow \mathbf{S}\, k}$$

$$\text{S-SUCC-CONG}\quad \frac{e \rightsquigarrow e'}{\mathbf{succ}\, e \rightsquigarrow \mathbf{succ}\, e'}$$

$$\text{S-NREC-ZERO}\quad \frac{}{\mathbf{nrec}\, 0\, \mathbf{of}\, \{0 \Rightarrow e_1;\ \mathbf{S}\, x \Rightarrow e_2\} \rightsquigarrow e_1}$$

$$\text{S-NREC-SUCC}\quad \frac{}{\mathbf{nrec}\, (\mathbf{S}\, k)\, \mathbf{of}\, \{0 \Rightarrow e_1;\ \mathbf{S}\, x \Rightarrow e_2\} \rightsquigarrow (e_2[k/x])\ (\mathbf{nrec}\, k\, \mathbf{of}\, \{0 \Rightarrow e_1;\ \mathbf{S}\, x \Rightarrow e_2\})}$$

$$\text{S-NREC-CONG}\quad \frac{e \rightsquigarrow e'}{\mathbf{nrec}\, e\, \mathbf{of}\, \{0 \Rightarrow e_1;\ \mathbf{S}\, x \Rightarrow e_2\} \rightsquigarrow \mathbf{nrec}\, e'\, \mathbf{of}\, \{0 \Rightarrow e_1;\ \mathbf{S}\, x \Rightarrow e_2\}}$$

Figure 2.1: Natural number operations: successor and recursion

number, with this operation, we can compute the successor of any expression that evaluates to a number.

Second, we add a *primitive recursion* operator to STLC. This extension allows us to define STLC programs by recursion. This form of recursion will always be bounded; we will be able to write interesting computations but will be able to know that all expressions terminate.

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

The successor operation $\mathbf{succ}\, e$, 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 $\mathbf{succ}\, e$ of the expression language, with the syntax $\mathbf{S}\, k$ 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 $\mathbf{S}\, k$ 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.

2.1 Examples

Let's use these operation!

double First, we define a doubling function on natural numbers with the following definition.

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

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

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

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.

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

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

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

2.2 Exercises

Proof assistants, such as Rocq, have built-in natural number types with induction and recursion principles. To become more familiar with using **nrec**, you can try defining operations in Rocq using its primitive recursion principles.

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

2.3 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 Gödel's System T [G5̈8], which was designed to study the consistency of arithmetic. The terminology that we use for "primitive recursion" is not quite the same as the related concept from computability theory. In that context, the natural number recursor is restricted to produce functions with types of the form $\mathbf{Nat} \to \mathbf{Nat} \to \ldots \to \mathbf{Nat}$, 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 [Men87]).

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)*

$$\text{BS-VAL}\ \frac{}{v \Rightarrow v} \qquad \text{BS-APP}\ \frac{e_1 \Rightarrow \lambda x. e_1' \qquad e_2 \Rightarrow v_1 \qquad e_1'[v_1/x] \Rightarrow v_2}{e_1\ e_2 \Rightarrow v_2}$$

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 $e \rightsquigarrow^* v$ 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 (Step expansion)**.** If $e \rightsquigarrow e'$ 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)**.** $v \rightsquigarrow^* v$

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

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 $e_1 \rightsquigarrow^* e_1'$ then $e_1\ e_2 \rightsquigarrow^* e_1'\ e_2$.

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

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. But, note that while the relation $e \Rightarrow v$ is deterministic, it represents a *partial function*. There are many expressions e that are not related to values.

This partiality leads to a significant drawback of a big-step semantics: it handles both the partiality of a runtime type error and the partiality of divergence in exactly the same way. This is in contrast to the small-step semantics: runtime errors made the single-step relation partial, while divergence can be modeled using a coinductive definition of multi-step reduction.

Because the big step semantics does not distinguish runtime errors from diverging programs, we run into difficulty when stating and proving type safety. While preservation holds for this semantics:

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

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

Furthermore, it is tempting to define type safety as follows:

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

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.

3.3 Further Reading

Kahn [Kah87] initially proposed the use of *natural semantics*, which has since been referred to as *big-step* semantics to contrast with *small-step* semantics.

Leroy [Ler06] explores the rammifications of using coinductive definitions for big-step semantics.

Charguéraud develops a compromise between big-step semantics and small-step semantics that he calls *pretty-big-step semantics* [Cha13].

4

Big-step termination and Semantic soundness

In this chapter we will show how to prove the strong version of type safety for a language with a big-step semantics. After completing the proof, not only will we know that programs do not crash, we will also know that they do not diverge either. Every well-typed program can produce a value.

4.1 Big-step preservation

Before we start with type safety, let's consider the preservation and progress lemmas from the small step relation.

Lemma 4.1.1 (Preservation)**.** If $\vdash e \in \tau$ and $e \Rightarrow v$ then $\vdash v \in \tau$.

Proof. Proof by induction on $\vdash e \in \tau$.

- For the literal case, e is some number k that steps to itself. It already has type $\mathbf{Nat}$.
- For the variable case, we assume the term is closed, so this case is impossible.
- For the abstraction case, e is some well-typed lambda expression $\lambda x.e$ that steps to itself.
- For the application case, e is $e_1 \;\; e_2$ where $\vdash \; e_1 \; \in \; \tau_1 \; \rightarrow \; \tau$ and $\vdash \; e_2 \; \in \; \tau_1$. By inversion on the evaluation relation, we also know that $e_1 \Rightarrow \lambda x.e_1'$ and $e_2 \Rightarrow v_2$ and $e_1'[v_2/x] \Rightarrow v$. By induction, we know that $\vdash \lambda x.e_1' \in \tau_1 \rightarrow \tau$ and $\vdash v_2 \in \tau_1$. By inverting the former, we also know that $x : \tau_1 \vdash e_1' \in \tau$. This means that we can use our substitution lemma to conclude $\vdash e_1'[v_2/x] \in \tau$.

□

The preservation lemma goes through readily. However, by itself, it isn't a type safety theorem because it does not give use any confidence that the type system rules out run-time errors. The lemma is stated so that we can only make a conclusion when we already have a successful run to a value. But, even if a term type checks, we can't tell whether it will produce anything.

The small step semantics had a progress lemma that reassured us that good things would happen if a program type checks — that program would make some incremental progress. But with this semantics we do not have a notion of incremental progress. So there isn't a ready analogue of this lemma available for us.

4.2 Big Step Safety

Instead, consider this statement of type safety for the big-step semantics. It states that any closed, well-typed term e must evaluate to some value v. This means that the term cannot crash or diverge.[1]

Conjecture 4.2.1 (Big Step Safety)**.** If $\vdash e \in \tau$ then $e \Rightarrow v$.

We won't be able to prove this theorem directly by induction on e or on the typing derivation. But, looking at how this proof fails will tell us how to strengthen it so that it is provable. We will do this strengthening incrementally, generalizing the induction hypothesis as necessary.

Our first attempt to prove the big step safety theorem is by induction on e.

- For the literal case, we know that e is a constant k (which is a value) because it is well typed, then τ is the type $\mathbf{Nat}$. Immediately we have $k \Rightarrow k$.
- Similarly for the abstraction case, we know that e is $\lambda x.e'$ and that τ is $\tau_1 \to \tau_2$. Again, immediately, we have $\lambda x.e' \Rightarrow \lambda x.e'$.
- We don't need to consider the variable case as the term is closed.
- Finally, consider the application case, where e is $e_1\ e_2$ and by inversion on the typing derivation we know that $\vdash e_1 \in \tau_1 \to \tau$ and $\vdash e_2 \in \tau_1$. We can use induction on e_1 and e_2 to get $e_1 \Rightarrow v_1$ and $e_2 \Rightarrow v_2$. Furthermore, preservation tells us that $\vdash v_1 \in \tau_1 \to \tau$, and we also know (via canonical forms) that the only closed values with function types are functions, so v_1 must be some $\lambda x.e_1'$. However, here we are stuck. We want to show that there is some v_3 such that $e_1'[v_2/x] \Rightarrow v_3$. But we don't have a way to conclude this. We can't use induction because $e_1'[v_2/x]$ isn't a subterm of e.

The solution at this point is to observe that our induction hypothesis was not strong enough. We needed to know more about e_1 than just that it terminates. We also need to know that when given any argument, *the application will also terminate*.

Therefore, let's strengthen the property that we prove. Above, in the application case we want to know something about $e_1'[v_2/x]$, so we cannot use the induction hypothesis on this term. However, we get this term from evaluating e_1 and e_2, so we can strengthen the conclusion of our lemma. Above, all we know about v is that it is a value. But, depending on this type, we can assert stronger properties about this value.

[1]For simplicity, we will work with a version of type safety that is a little weaker than the version in the previous chapter; it doesn't require the result to be well typed.

Definition 4.2.1 (Value set)**.** Define the family of sets $\mathcal{V}[\![\tau]\!]$ by structural recursion on τ.

$$\begin{array}{lcl} \mathcal{V}[\![\mathbf{Nat}]\!] & = & \mathbb{N} \\ \mathcal{V}[\![\tau_1 \to \tau_2]\!] & = & \{\lambda x.e \mid \forall v,\ v \in \mathcal{V}[\![\tau_1]\!] \textit{ implies } \exists v', e[v/x] \Rightarrow v' \textit{ and } v' \in \mathcal{V}[\![\tau_2]\!]\} \end{array}$$

This definition interprets each type as a set of values. The values in each set make sense for the associated type. The set for **Nat** contains all natural numbers, and the sets for each function type only contain abstractions. These abstractions must themselves act like they have the right type: they must take any value from the interpretation of the argument type and, after substitution, evaluate to a value in that is in the result type. It is this requirement for function types that will help us complete the proof above—we know that the substitution we need $e'[v_2/x]$ will terminate (and satisfy the condition of our stronger theorem).

This idea of evaluating a term to a value in a particular set will come up again, so let's name it. The computational interpretation of a type includes all expressions that evaluate to a value in the value set for that type.

Definition 4.2.2 (Computation set)**.**

$$\mathcal{C}[\![\tau]\!] = \{\ e \mid e \Rightarrow v \textit{ and } \quad v \in \mathcal{V}[\![\tau]\!]\ \}$$

Now we can use these definitions to strengthen our safety theorem. Not only do we require that a term evaluate to a value, but that value must be in the appropriate set for that type.

Lemma 4.2.1 (Semantic soundness)**.** If $\vdash\ e \in \tau$ then $e \in \mathcal{C}[\![\tau]\!]$.

However, as you might guess, our induction hypothesis is *still* not strong enough to prove the result. Let's try a case where it works, and one where it doesn't.

Let's prove the theorem by induction on $\vdash\ e \in \tau$. The variable case is impossible and the literal case is immediate. Now consider the application case, where we have $\vdash\ e_1\ e_2 \in \tau$, with $\vdash\ e_1 \in \tau_1 \to \tau$ and $\vdash\ e_2 \in \tau_1$. By induction, we know that $e_1 \in \mathcal{C}[\![\tau_1 \to \tau]\!]$ and $e_2 \in \mathcal{C}[\![\tau_1]\!]$. So that means that both of these terms must evaluate to values in their appropriate sets. i.e. $e_1 \Rightarrow v_1$ where $v_1 \in \mathcal{V}[\![\tau_1 \to \tau]\!]$ and $e_2 \Rightarrow v_2$ where $v_2 \in \mathcal{V}[\![\tau_1]\!]$. To construct an evaluation in this case, we need to know that v_1 is some abstraction, and that if we substitute v_2 for the parameter the program terminates. But this is exactly what the set $\mathcal{V}[\![\tau_1 \to \tau]\!]$ gives us! We have fixed this case.

However, now consider the application case. In this case we have $\vdash\ \lambda x.e \in \tau_1 \to \tau_2$. We need to show that this abstraction is in $\mathcal{C}[\![\tau_1 \to \tau_2]\!]$. We know already that this term has the right type. It also terminates as it is already a value; i.e. $\lambda x.e \Rightarrow \lambda x.e$. Our remaining goal is to show $\lambda x.e \in \mathcal{V}[\![\tau_1 \to \tau_2]\!]$. But, we cannot make any more progress. We don't have an induction hypothesis that we can use in this case, because our lemma only applies to closed terms.

Therefore, we will strengthen our lemma *again*, so that it applies not just to closed terms, but also gives us an induction hypothesis for open terms. But, our sets are only sets of closed terms! How can we do this?

The answer is that our revised lemma should quantify over *closing substitutions*, i.e. substitutions that replace all free variables with closed values. Furthermore, to make sure that our theorem is strong enough, we also require that these closed values be part of our semantic sets.

Definition 4.2.3 (Semantic substitution)**.** Define $\sigma \in \mathcal{G}[\![\Gamma]\!]$ when $\forall x \in \mathsf{dom}\,\Gamma, \sigma\, x \in \mathcal{V}[\![(\Gamma\, x)]\!]$.

With this definition, we can now restate our lemma in its final form.

Lemma 4.2.2 (Semantic soundness)**.** If $\Gamma \vdash e \in \tau$ then forall $\sigma \in \mathcal{G}[\![\Gamma]\!]$, $e[\sigma] \in \mathcal{C}[\![\tau]\!]$.

Proof. Proof is by induction on the derivation of $\Gamma \vdash e \in \tau$.

- If $\Gamma \vdash x \in \Gamma\, x$, then given an arbitrary $\sigma \in \mathcal{G}[\![\Gamma]\!]$, we need to show that $x[\sigma] \in \mathcal{C}[\![(\Gamma\, x)]\!]$. However, in our definition of substitution, we have $x[\sigma] = \sigma\, x$, so we know that $\sigma\, x \in \mathcal{V}[\![(\Gamma\, x)]\!]$ by assumption. This also implies that $\sigma\, x \in \mathcal{C}[\![(\Gamma\, x)]\!]$ because values evaluate to themselves.

- If $\Gamma \vdash k \in \mathbf{Nat}$, then given an arbitrary $\sigma \in \mathcal{G}[\![\Gamma]\!]$, we need to show that $k[\sigma] \in \mathcal{C}[\![\mathbf{Nat}]\!]$. This is the same as showing that $k \Rightarrow k$ and $k \in \mathbb{N}$.

- Say $\Gamma \vdash e_1\ e_2 \in \tau$ where $\Gamma \vdash e_1 \in \tau_1 \to \tau$ and $\Gamma \vdash e_2 \in \tau_1$ are subderivations. Given an arbitrary $\sigma \in \mathcal{G}[\![\Gamma]\!]$, we need to show that $(e_1\ e_2)[\sigma] \in \mathcal{C}[\![\tau]\!]$. By induction we know that $e_1[\sigma] \in \mathcal{C}[\![\tau_1 \to \tau]\!]$ and $e_2[\sigma] \in \mathcal{C}[\![\tau_1]\!]$. This means that the former steps to some $v_1 \in \mathcal{V}[\![\tau_1 \to \tau]\!]$ and the latter steps to some $v_2 \in \mathcal{V}[\![\tau_1]\!]$. Furthermore, we know that v_1 must be some λ-term $\lambda x.e$, and that $e[v_2/x] \Rightarrow v$ with $v \in \mathcal{V}[\![\tau]\!]$. But, by rule BS-APP, we also have $e_1[\sigma]\ e_2[\sigma] \Rightarrow v$, which gives us our goal.

- Say $\Gamma \vdash \lambda x.e \in \tau_1 \to \tau$ where $\Gamma, x : \tau_1 \vdash e \in \tau$. Given an arbitrary $\sigma \in \mathcal{G}[\![\Gamma]\!]$, we need to show that $(\lambda x.e)[\sigma] \in \mathcal{C}[\![\tau]\!]$. We know that $(\lambda x.e)[\sigma] = \lambda x.e[x/x, \sigma]$ (assuming that x is not in the domain or range of σ). By our substitution lemma, we know that $\vdash (\lambda x.e)[\sigma] \in \tau_1 \to \tau_2$. It is a value, so it steps to itself. To show that this term is in the appropriate value set, we need to assume some $v_1 \in \mathcal{V}[\![\tau_1]\!]$ and show that $e[x/x, \sigma][v_1/x] \in \mathcal{V}[\![\tau]\!]$. Substitutions compose, so this is the same as saying $e[v_1/x, \sigma] \in \mathcal{V}[\![\tau]\!]$. This goal follows by our induction hypothesis as long as $v_1/x, \sigma \in \mathcal{G}[\![\Gamma, x{:}\tau]\!]$. But this follows by definition because $\sigma \in \mathcal{G}[\![\Gamma]\!]$ and $v_1 \in \mathcal{V}[\![\tau]\!]$.

□

After strengthening our lemma twice, we have put it in a form that we can prove. Furthermore, this form implies the original version, so we have proven type safety.

4.3 Rephrasing semantic soundness

Where does the name *semantic soundness* come from in this lemma? What is semantic about this argument?

The type safety proofs we have seen so far are syntactic: they involve working with the syntax of programs, typing derivations (which are syntactic objects), substitution and rewriting relations (which is a syntactic manipulation). Semantic interpretations of the λ-calculus give us a more mathematical meaning for our programs. And one idea in semantics is to give a meaning to types based on sets of values.

Furthermore, some authors like to define $\mathcal{V}[\![\tau]\!]$ by first defining operations on sets.

Definition 4.3.1 (Function set)**.** Define $T_1 \Rightarrow T_2$, which constructs a set of terms from two given sets of terms T_1 and T_2 as follows:

$$T_1 \Rightarrow T_2 = \{\, \lambda x.e_2 \,|\, \textit{for all} \;\; e_1, \quad e_1 \in T_1 \textit{ implies } e_2[e_1/x] \in T_2 \,\}$$

With this definition, we can define our value set using this operation:

$$\begin{array}{lcl} \mathcal{V}[\![\mathbf{Nat}]\!] & = & \mathbb{N} \\ \mathcal{V}[\![\tau_1 \to \tau_2]\!] & = & \mathcal{V}[\![\tau_1]\!] \Rightarrow \mathcal{C}[\![\tau_2]\!] \end{array}$$

Furthermore, we can also define a notion for "semantic typing": the idea that a term is well typed semantically when given any closing substitution, it is an element of the appropriate substitution set.

Definition 4.3.2 (Semantic typing)**.** Define $\Gamma \vDash e : \tau$ when forall $\sigma \in \mathcal{G}[\![\Gamma]\!]$, $e[\sigma] \in \mathcal{C}[\![\tau]\!]$.

With this definition, we can prove semantic typing rules that are analogous to each syntactic typing rule. The proofs of these lemmas are the subcases of the semantic soundness proof.

Lemma 4.3.1 (Semantic var rule)**.** $\Gamma \vDash x : \Gamma\, x$.

Lemma 4.3.2 (Semantic lit rule)**.** $\Gamma \vDash k : \mathbf{Nat}$.

Lemma 4.3.3 (Semantic abs rule)**.** If $\Gamma, x\!:\!\tau \vDash e : \tau_1 \to \tau$, then $\Gamma \vDash \lambda x.e : \tau_1 \to \tau$.

Lemma 4.3.4 (Semantic app rule)**.** If $\Gamma \vDash e_1 : \tau_1 \to \tau_2$ and $\Gamma \vDash e_2 : \tau_1$, then $\Gamma \vDash e_1\ e_2 : \tau_2$.

These lemmas allow us to rephrase the semantic soundness theorem nicely.

Theorem 4.3.1 (Semantic typing)**.** If $\Gamma \vdash e \in \tau$ then $\Gamma \vDash e : \tau$

Proof. Induction on the typing derivation, applying the appropriate semantic typing rule in each case. □

4.4 Variations

This proof is usually stated with a small-step semantics and tutorials are available from a variety of references. However, with a small-step semantics you need an additional lemma: closure under reverse evaluation. Terms that step to terms in the computation sets are themselves in the computation set.

Lemma 4.4.1 (Closure under expansion)**.** If $e \rightsquigarrow e'$ and $e' \in \mathcal{C}[\![\tau]\!]$ then $e \in \mathcal{C}[\![\tau]\!]$.

Not all authors make a distinction between value sets $\mathcal{V}[\![\tau]\!]$ and computation sets $\mathcal{C}[\![\tau]\!]$, combining them together into a uniform definition.

You might also ask whether semantic *completeness* holds.

Lemma 4.4.2 (Semantic completeness)**.** If $\Gamma \vDash e : \tau$ then $\Gamma \vdash e \in \tau$.

In other words, do our sets of terms contain only well-typed terms? The answer is no! A program can include any sort of stuck subterm, as long as that subterm is never executed. For example, consider $(\lambda x.3)\ (\lambda y.1\ 0)$. This term doesn't type check because it has the stuck application $1\ 0$ as a subterm. But it evaluates to 3 which is in $\mathcal{V}[\![\mathbf{Nat}]\!]$, so we can say that $\vDash (\lambda x.3)\ (\lambda y.1\ 0) : \mathbf{Nat}$.

The reason that semantic completeness fails, is that we include ill-typed terms in our sets as long as they are harmless. But, it you believe that only typed terms are worth discussing, you might prefer to restrict the sets to well-typed terms. In that case, completeness holds trivially.

4.5 Further reading

Robert Harper's note: *How to (Re)Invent Tait's Method*https://www.cs.cmu.edu/~rwh/courses/chtt/pdfs/tait.pdf includes this proof for a small-step semantics of STLC with booleans, unit, products and functions. Harper calls the computation sets "hereditarily terminating."

This proof is the simplest example of a general proof technique called *proof by logical relations*. The set $\mathcal{V}[\![\tau]\!]$ is a *unary* logical relation, also called a logical predicate. Binary logical relations can be used to show program equivalence or noninterference. Or, extending the sets to include open terms means that we can show that full reduction for STLC (even inside functions) always terminates. However, these proofs take a few more steps than the one that we

5

REC: Recursive definitions

In this chapter, we introduce our first *effectful* programming language: a fine-grained call-by-value language with recursive definitions, called REC. That effect is *nontermination*, where evaluating a program may not actually result in a value.

We will add nontermination through two sorts of recursive definitions: first through recursive values and then through recursive types.

5.1 Recursive definitions in CBV languages

To build intuition about the language structures we are adding in this section, we will first start out with some examples written in the OCaml programming language.

For example, a straightforward definition of the doubling function, which we previously implemented via primitive recursion, looks like this:

```
let rec double : nat -> nat = fun x ->
  match x with
  | O -> O
  | S y -> succ (succ (double y))
```

However, we are not limited to primitive recursion in OCaml. We can do much more.

5.1.1 Mutual recursion via recursive tuples

First, OCaml allows the definition of functions that are *mutually* recursive using the keywords `rec ... and`. Any definition in a block can refer to any other in the same block.

A simple example is the mutual definition of the odd and even functions.

```
let rec even : nat -> bool = fun x ->
  match x with
  | O -> true
  | S y -> odd y
```

```
  and
  odd : nat -> bool = fun x ->
  match x with
  | O -> false
  | S y -> even y
```

But what if we didn't have **and** available? It turns out that there are two ways that we could replace the code above.

One way is to inline one recursive definition into another. This requires us to repeat the definition of one of the operations later.

```
let rec even : nat -> bool = fun x ->
  match x with
  | O -> true
  | S y -> match y with
           | O -> false
           | S z -> even z

let odd : nat -> bool = fun x ->
  match x with
  | O -> false
  | S y -> even y
```

Bekić's theorem[1] states that this always works.

Another option is to use recursion on products of functions, instead of just on functions. In this version, to make the code a little easier to read, we first define a record type containing odd and even functions (i.e. a product). Then, by using recursion through the product value, we can mutually define the two functions.

```
type oe = { odd : nat -> bool ; even : nat -> bool }

let rec obj : oe = { odd = (fun x ->
                              match x with
                              | O -> false
                              | S y -> obj.even y) ;
                     even = (fun x ->
                              match x with
                              | O -> true
                              | S y -> obj.odd y
                            )
                   }
```

5.1.2 Another example with recursive tuples

However, it turns out that in OCaml there are definitions of recursive product values where the recursive reference is not hidden in a function body.

Consider the following stream type:

```
type stream = Cons of int * stream

let rec zeros : stream = Cons (0, zeros)
```

[1] https://en.wikipedia.org/wiki/Beki%C4%87%27s_theorem

Perhaps surprisingly OCaml accepts the definition of `zeros` above. This line defines `zeros` as a recursive value, represented by a cycle in the heap, and displayed as `Cons (0, <cycle>)`.

We are limited with what we can do with the `stream` type. For example, although we define a mapping operation, thus:

```
let rec map_stream f = fun y ->
  match y with
  | Cons (x, s) -> Cons (f x, map_stream f s)
```

any use of the mapping function will go into an infinite loop:

```
let ones : stream = map_stream (fun x -> x + 1) zeros    (* DIVERGES! *)
```

We cannot use the stream for termination. Instead, we need to find some other finite value to iterate over.

For example, we can access any finite number of zeros from the stream:

```
let rec take : nat -> stream -> int list * stream = fun n s ->
  match n with
  | O -> ([],s)
  | S m -> match s with
           | Cons (x, s') ->
              let (tl, s'') = take m s' in (x :: tl, s'')

let (example,_)  = take n5 zeros
```

But there is a natural number that we could give to **`take`** that would cause it to diverge.

```
let rec omega : nat = S omega
let example = take omega zeros (* DIVERGES *)
```

5.1.3 Recursive values via recursive types

So far, we have been showing examples that use recursive value definitions as well as recursive types. But, it turns out that we only need the latter to define the former. At least for recursive functions. The trick is that we can use a recursive type allow the Y-combinator to type check in OCaml.

Let's use a simple **`length`** function as an example.

```
let rec length : int list -> int = fun l ->
    match l with
    | [] -> 0
    | _ :: l' -> 1 + length l'
```

If we knew how big of list we wanted to call the length function on, then we would not need to use recursion for this definition. For example, say we only needed it to work for lists of size 0 or 1. Then we can use the definition of `length2` below:

```
let bottom : int list -> int =
  fun _ -> failwith "<loops>"

let length0 : int list -> int = bottom
```

```
let length1 : int list -> int = fun l ->
  match l with
  | [] -> 0
  | _ :: l' -> 1 + length0 l'

let length2 : int list -> int = fun l ->
  match l with
  | [] -> 0
  | _ :: l' -> 1 + length1 l'
```

When given a longer list, this function fails. But, this is a pretty long definition. Let's refactor it:

The refactored version is easy to see how to extend it to work with longer lists.

```
let length4 : int list -> int =
  (fun f -> (f (f (f (f bottom)))))
  mk_length
```

More generally, our recursive definition is

```
let rec length : int list -> int =
  fun l ->
    mk_length length
    l
```

which could also be written as:

```
let rec length : int list -> int =
  fun l ->
    mk_length (fun l -> mk_length length l)
    l
```

Here's the big leap! If we could do self application, then maybe we would write this example without using `rec`.

```
(* DOESN'T TYPE CHECK *)
let length : int list -> int =
   fun l ->
     (fun f -> f f)
     (fun g -> mk_length (fun l -> g g l))
     l
```

But this code does not type check in OCaml.

To get it to type check, we need to introduce a recursive type so that we can apply a function to itself. The key part of the type definition is that recursive reference occurs to the left of the arrow in its own definition. This is called a negative occurrence and would be rejected by Rocq, but it is not an inductive type.

```
type 'a dom = Abs of ('a dom -> 'a)

let app (f : 'a dom) (x : 'a dom) : 'a =
  match f with
  | Abs h -> h x
```

Indeed, with this type, we can write the simplest infinite loop in the untyped lambda-calculus. This term steps immediately to itself.

```
let rec f : int -> void =
  fun x -> f x

let loop : void = f 0
```

Putting this idea to work, we can use this type get the length function above to type check.

```
let length : int list -> int =
   fun l ->
     (fun f -> app f f)
     (Abs (fun g ->
         mk_length (fun l -> app g g l)))
     l
```

We can also factor out the call-by-value Y-combinator, which is the essence of the recursive definition.

```
let y_combinator : (('a -> 'b) -> ('a -> 'b)) -> ('a -> 'b)
  = fun h ->
   (fun f -> app f f)
   (Abs (fun g -> h (fun x -> (app g g) x)))

let length = y_combinator mk_length
```

5.2 A fine-grained CBV language

Now, before we consider the semantics of recursive definitions, let's refactor the base language to make it easier to extend. At the same time, we will add a few more common language features, such as products, sums and an empty type.

Here is the syntax of the base language that we will work with.

τ	$::=$	$\mathbf{Void} \mid \mathbf{Nat} \mid \tau_1 \to \tau_2 \mid \tau_1 * \tau_2 \mid \tau_1 + \tau_2$	types
v	$::=$	x	Variables
	$\mid$	k	Natural numbers
	$\mid$	$\lambda x.e$	Functions
	$\mid$	(v_1, v_2)	Pairs
	$\mid$	$\mathbf{inj}_1\, v \mid \mathbf{inj}_2\, v$	Sums
e	$::=$	$v_1\ v_2$	application
	$\mid$	$\mathbf{succ}\ v$	successor
	$\mid$	$\mathbf{case}\ v\ \mathbf{of}\ \{0 \Rightarrow e_1;\ \mathbf{S}\ x \Rightarrow e_2\}$	test for zero
	$\mid$	$\mathbf{prj}_1\, v \mid \mathbf{prj}_2\, v$	projection
	$\mid$	$\mathbf{case}\ v\ \mathbf{of}\ \{\mathbf{inj}_1 x \Rightarrow e_1; \mathbf{inj}_2 x \Rightarrow e_2\}$	case
	$\mid$	$\mathbf{ret}\ v$	value
	$\mid$	$\mathbf{let}\ x\ = e_1\ \mathbf{in}\ e_2$	sequencing

The first thing to notice about this language is that it makes a syntactic distinction between values and expressions. These two grammars are mutually defined.

Terms appear inside values in function bodies. Values can appear in terms in several ways. First by being "returned" via $\mathbf{ret}\ v$. (Some versions of fine-grained CBV make the $\mathbf{ret}$ implicit, as it often can be inferred from context. For clarity, we will make it explicit.)

Furthermore, notice that almost all terms restrict their "active" subterms to be values. The arguments of $\mathbf{succ}\ v$, the scrutinee for the various forms of pattern matching, and both the function and the argument in an application must be values.

There are several reasons for this modification. The one that we will discuss now is that it allows us to extend the small-step semantics with new constructs while skipping the associated congruence rules in the operational semantics. All we need to add are the main computation rules. As the active part of the expression must be a value, we do not need to add any rules to evaluate that subexpression.

Definition 5.2.1 (Active rules). $\boxed{e \rightsquigarrow e'}$ *(term e steps to e')*

S-BETA
$$\frac{}{(\lambda x.e)\ v \rightsquigarrow e[v/x]}$$

S-SUCC
$$\frac{}{\mathbf{succ}\ k \rightsquigarrow \mathbf{S}\ k}$$

S-CASE-ZERO
$$\frac{}{\mathbf{case}\ 0\ \mathbf{of}\ \{0 \Rightarrow e_1;\ \mathbf{S}\ x \Rightarrow e_2\} \rightsquigarrow e_1}$$

S-CASE-SUCC
$$\frac{}{\mathbf{case}\ (\mathbf{S}\ k)\ \mathbf{of}\ \{0 \Rightarrow e_1;\ \mathbf{S}\ x \Rightarrow e_2\} \rightsquigarrow e_2[k/x]}$$

S-PRJ1
$$\frac{}{\mathbf{prj}_1(v_1, v_2) \rightsquigarrow \mathbf{ret}\ v_1}$$

S-PRJ2
$$\frac{}{\mathbf{prj}_2(v_1, v_2) \rightsquigarrow \mathbf{ret}\ v_2}$$

S-CASE-INJ1
$$\frac{}{\mathbf{case}\ (\mathbf{inj}_1 v)\ \mathbf{of}\ \{\mathbf{inj}_1 x \Rightarrow e_1; \mathbf{inj}_2 x \Rightarrow e_2\} \rightsquigarrow e_1[v/x]}$$

S-CASE-INJ2
$$\frac{}{\mathbf{case}\ (\mathbf{inj}_2 v)\ \mathbf{of}\ \{\mathbf{inj}_1 x \Rightarrow e_1; \mathbf{inj}_2 x \Rightarrow e_2\} \rightsquigarrow e_2[v/x]}$$

In this language, the let term controls the sequencing of evaluation. We only have a single congruence rule, which evaluates right-hand side of a let expression if it is not a returned value.

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

S-LET-RET
$$\frac{}{\mathbf{let}\ x\ = \mathbf{ret}\ v\ \mathbf{in}\ e \rightsquigarrow e[v/x]}$$

S-LET-CONG
$$\frac{e_1 \rightsquigarrow e_1'}{\mathbf{let}\ x\ = e_1\ \mathbf{in}\ e_2 \rightsquigarrow \mathbf{let}\ x\ = e_1'\ \mathbf{in}\ e_2}$$

In this language, there is no question about the ordering in which evaluation happens. In some CBV languages, we might evaluate the argument of an application before we evaluate the function. Or we might evaluate the function before the argument. Or, the language might say that this order is undefined, giving flexibility to the language implementation. For example, the OCaml byte code compiler chooses a different order than the native code compiler.

In STLC, the order doesn't matter, and either version will produce the same result. However, in some effectful languages, the order of this evaluation can matter.

Therefore, to rule out ambiguity, this language puts the onus on the programmer to explicitly say what they want, by writing either

$$\mathbf{let}\ x_1\ = e_1\ \mathbf{in}\ \mathbf{let}\ x_2\ = e_2\ \mathbf{in}\ x_1\ x_2 \qquad \text{or} \qquad \mathbf{let}\ x_2\ = e_2\ \mathbf{in}\ \mathbf{let}\ x_1\ = e_1\ \mathbf{in}\ x_1\ x_2$$

5.2.1 Type system

With a syntactic distinction between values and expressions, we have two separate typing judgments: one for values and one for terms.

Definition 5.2.2 (Type system).

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

$$\frac{}{\Gamma \vdash k \in \mathbf{Nat}}\ \textsc{TV-Lit} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x \in \tau}\ \textsc{TV-Var} \qquad \frac{\Gamma, x{:}\tau_1 \vdash e \in \tau_2}{\Gamma \vdash \lambda x.e \in \tau_1 \to \tau_2}\ \textsc{TV-Abs} \qquad \frac{\Gamma \vdash v_1 \in \tau_1 \quad \Gamma \vdash v_2 \in \tau_2}{\Gamma \vdash (v_1, v_2) \in \tau_1 * \tau_2}\ \textsc{TV-Pair}$$

$$\frac{\Gamma \vdash v_1 \in \tau_1}{\Gamma \vdash \mathbf{inj}_1 v_1 \in \tau_1 + \tau_2}\ \textsc{TV-Inj1} \qquad \frac{\Gamma \vdash v_2 \in \tau_2}{\Gamma \vdash \mathbf{inj}_2 v_2 \in \tau_1 + \tau_2}\ \textsc{TV-Inj2}$$

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

$$\frac{\Gamma \vdash v \in \tau}{\Gamma \vdash \mathbf{ret}\ v \in \tau}\ \textsc{TE-Ret} \qquad \frac{\Gamma \vdash e_1 \in \tau_1 \quad \Gamma, x{:}\tau_1 \vdash e_2 \in \tau}{\Gamma \vdash \mathbf{let}\ x\ = e_1\ \mathbf{in}\ e_2 \in \tau}\ \textsc{TE-Let} \qquad \frac{\Gamma \vdash v_1 \in \tau_1 \to \tau_2 \quad \Gamma \vdash v_2 \in \tau_1}{\Gamma \vdash v_1\ v_2 \in \tau_2}\ \textsc{TE-App}$$

$$\frac{\Gamma \vdash v \in \tau_1 * \tau_2}{\Gamma \vdash \mathbf{prj}_1 v \in \tau_1}\ \textsc{TE-Prj1} \qquad \frac{\Gamma \vdash v \in \tau_1 * \tau_2}{\Gamma \vdash \mathbf{prj}_2 v \in \tau_2}\ \textsc{TE-Prj2}$$

$$\frac{\Gamma \vdash v \in \tau_1 + \tau_2 \quad \Gamma, x_1{:}\tau_1 \vdash e_1 \in \tau \quad \Gamma, x_2{:}\tau_2 \vdash e_2 \in \tau}{\Gamma \vdash \mathbf{case}\ v\ \mathbf{of}\ \{\mathbf{inj}_1 x_1 \Rightarrow e_1; \mathbf{inj}_2 x_2 \Rightarrow e_2\} \in \tau}\ \textsc{TE-Case}$$

These two judgments are mutually defined.

5.2.2 Recovering expressiveness

Note that this restriction does not limit the expressiveness of the language. We can *define* the standard expression forms.

For example, instead of application of the form $e_1\ e_2$, the only application form in this language is $v_1\ v_2$.

Definition 5.2.3 (Eager let). Define **let** $x \Leftarrow e_1$ **in** e_2 as $e_2[v/x]$ when e_1 is **ret** v and **let** $x = e_1$ **in** e_2 otherwise.

We use this eager let in the definition of some derived forms.

Definition 5.2.4 (Extended Application). Define $e_1\ e_2$ as **let** $x_1 \Leftarrow e_1$ **in let** $x_2 \Leftarrow e_2$ **in** $x_1\ x_2$.

Definition 5.2.5 (Extended Successor)**.** Define $\mathbf{succ}\, e$ as $\mathbf{let}\ x \Leftarrow e\ \mathbf{in}\ \mathbf{succ}\, x$.

The reason for the eager let is so that we can prove that our derived forms have the appropriate operational behavior.

Lemma 5.2.1 (Application steps)**.**

1. $(\mathbf{ret}\,(\lambda x.e))\ (\mathbf{ret}\ v) \rightsquigarrow e[v/x]$.
2. If $e_1 \rightsquigarrow e_1'$ then $e_1\ e_2 \rightsquigarrow^* e_1\ e_2'$.
3. If $e_2 \rightsquigarrow e_2'$ then $v\ e_2 \rightsquigarrow^* v\ e_2'$.

5.3 Recursive values

Now, let's add general recursion. We will do so by adding a new form of value, called a *recursive value*, written $\mathbf{rec}\, x.v$. Here, the variable x is bound inside v and refers to the whole value.

$$v ::= \mathbf{rec}\, x.v$$

Usually v will be a function, and this will give is a way to define recursive functions. But it doesn't have to be, as we saw in OCaml.

The typing rule includes an auxiliary judgment written $\tau\ \mathsf{ok}$ that specifies which types may be used in recursive definition.

$$\text{TV-REC}\quad \frac{\tau\ \mathsf{ok} \qquad \Gamma, x{:}\tau \vdash v \in \tau}{\Gamma \vdash \mathbf{rec}\, x.v \in \tau}$$

For now, the rules specify that function and product types are ok.

$$\text{FUN-OK}\quad \frac{}{(\tau_1 \rightarrow \tau_2)\ \mathsf{ok}} \qquad\qquad \text{PROD-OK}\quad \frac{}{(\tau_1 * \tau_2)\ \mathsf{ok}}$$

Analogously, we need small-step rules to unwind the value when that type is used.

$$\text{S-APP-REC}\quad \frac{}{(\mathbf{rec}\, x.v_1)\ v_2 \rightsquigarrow v_1[\mathbf{rec}\, x.v_1/x]\ v_2}$$

$$\text{S-PRJ1-REC}\quad \frac{}{\mathbf{prj}_1(\mathbf{rec}\, x.v) \rightsquigarrow \mathbf{prj}_1 v[\mathbf{rec}\, x.v/x]} \qquad\qquad \text{S-PRJ2-REC}\quad \frac{}{\mathbf{prj}_2(\mathbf{rec}\, x.v) \rightsquigarrow \mathbf{prj}_2 v[\mathbf{rec}\, x.v/x]}$$

Why do we do it this way? Because of our syntactic separation between values and computations! We can only substitute values for values.

5.4 Recursive types

Finally we add recursive types to the language. This means adding a recursive definition form μ and type variables to the syntax of types. And, in the syntax of terms, we introduce two coercions, for introducing and eliminating values with the recursive type.

$$\begin{array}{rcll} \tau & ::= & \alpha \mid \mu\alpha.\tau & \text{variables and recursive types} \\ v & ::= & \textbf{fold}\ v & \text{introduction form} \\ t & ::= & \textbf{unfold}\ v & \text{elimination form} \end{array}$$

Although we have type variables in our types, they are only used for recursion in this language. It will be an invariant of our type system that we only work with closed types. Furthermore, a recursive type $\mu\alpha.\tau$ is closed, if and only if its unfolding $\tau[\mu\alpha.\tau/\alpha]$ is also closed.

The introduction form creates a value with a recursive type and the elimination rule exposes its structure.

$$\frac{\Gamma \vdash v \in \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash \textbf{fold}\ v \in \mu\alpha.\tau}\ \text{TV-FOLD} \qquad \frac{\Gamma \vdash v \in \mu\alpha.\tau}{\Gamma \vdash \textbf{unfold}\ v \in \tau[\mu\alpha.\tau/\alpha]}\ \text{TE-UNFOLD}$$

The single new rule of the operational semantics removes the coercions.

$$\frac{}{\textbf{unfold}\ (\textbf{fold}\ v) \rightsquigarrow \textbf{ret}\ v}\ \text{S-UNFOLD}$$

5.4.1 Recursive type variations

The language we have presented so far includes strict recursive types, where only values can be given recursive types. An alternative form delays evaluation when the recursive value is defined.

$$\begin{array}{rcll} \tau & ::= & \alpha \mid \nu\alpha.\tau & \text{variables and recursive types} \\ v & ::= & \textbf{roll}\ e & \text{introduction form} \\ t & ::= & \textbf{unroll}\ v & \text{elimination form} \end{array}$$

$$\frac{\Gamma \vdash e \in \tau[\nu\alpha.\tau/\alpha]}{\Gamma \vdash \textbf{roll}\ e \in \nu\alpha.\tau}\ \text{TV-ROLL} \qquad \frac{\Gamma \vdash v \in \nu\alpha.\tau}{\Gamma \vdash \textbf{unroll}\ v \in \tau[\nu\alpha.\tau/\alpha]}\ \text{TE-UNROLL}$$

(s-unroll?)

The presence of **fold** v and **unfold** v introduction and elimination forms, means that language includes *iso*-recursive types. The types $\mu\alpha.\tau$ and $\tau[\mu\alpha.\tau/\alpha]$ are isomorphic, but not equal types. The introduction and elimination terms are coercions for the isomorphism. Iso-recursive types are the most straightforward way to add recursive types to a language, as it only requires α-equivalence for type equality.

Alternatively, some languages include a definitional type equivalence, i.e. an equivalence relation that states when two types are equal and is coarser than α-equality. In this case, the language includes some sort of conversion rule, giving terms any equivalent types. This conversion rule can make the metatheory of the language more complex to work with as often the type system is not syntax-directed.

5.5 Type safety

Let's consider type safety for REC. Notably, we will do so by proving preservation and progress with the small-step semantics. While it is possible to define a big-step semantics for this language, the fact that computations may diverge means that we have to be careful.

5.6 Further reading

Fine-grained CBV is from [LPT03].

The definition of *eager let* is from Forster et al. [FSSS19].

6

Step-indexing: Semantic type safety for REC

We proved a semantic soundness theorem for STLC. Can prove the same result for REC? Of course not. We know that we are doomed to failure as the semantic soundness theorem for STLC also proves that all programs terminate, but that is not the case for this language.

6.1 A failing proof: semantic sets

However, let's take a look at the proof because it first gives us a chance to redo the argument from before, but using a small-step semantics this time instead of a big-step semantics. It is also instructive to look at where our proof breaks down, so we can think about how to modify the statement of the theorem so that we can prove something that is true.

First, we need to define our semantic sets for our fine-grained CBV language. Our definition is similar to last time, with a few changes.

Definition 6.1.1 (Semantic sets)**.**

$$
\begin{array}{lcl}
\mathcal{C}[\![\tau]\!] & = & \{\ e \mid e \rightsquigarrow^* \mathbf{ret}\ v \ \mathit{and} \quad v \in \mathcal{V}[\![\tau]\!]\ \} \\
\\
\mathcal{V}[\![\mathbf{Nat}]\!] & = & \mathbb{N} \\
\mathcal{V}[\![\mathbf{Void}]\!] & = & \{\} \\
\mathcal{V}[\![\tau_1 \to \tau_2]\!] & = & \{\ v \mid \forall v_2,\ v_2 \in \mathcal{V}[\![\tau_1]\!] \ \mathit{implies}\ v\ v_2 \in \mathcal{C}[\![\tau_2]\!]\} \\
\mathcal{V}[\![\tau_1 * \tau_2]\!] & = & \{\ v \mid \mathbf{prj}_1 v \in \mathcal{C}[\![\tau_1]\!] \ \mathit{and}\ \mathbf{prj}_2 v \in \mathcal{C}[\![\tau_2]\!]\} \\
\mathcal{V}[\![\tau_1 + \tau_2]\!] & = & \{\ \mathbf{inj}_1 v \mid v_1 \in \mathcal{V}[\![\tau_1]\!]\} \cup \{\ \mathbf{inj}_2 v \mid v_2 \in \mathcal{V}[\![\tau_2]\!]\}
\end{array}
$$

First, we define the sets for computations and values mutually. That will allow us to use the definition of computation sets directly in the definition of the value sets, so we can localize our description of the evaluation.

Next, we have a few more types around, so we need more cases in our definition of the value sets. We now have cases for the empty type (there are no values of this type, so its semantics is an empty set), product types and sum types. These cases are in addition to the natural number and function type cases from last time.

One case that is missing from the definition is the case for recursive types. We might like to add a definition like the following: a "folded" value is in the set for a recursive type if the underlying value is in the set for the unfolded type.

$$\mathcal{V}[\![\mu\alpha.\tau]\!] = \{\mathbf{fold}\, v \mid v \in \mathcal{V}[\![\tau[\mu\alpha.\tau/\alpha]]\!]\}$$

However, we **cannot** add this case to our definition. The reason is that with this case, the definition is no longer well-founded. Above, the semantic sets are defined by recursion over the type structure. This means that each recursive part of the definition must be to a set for a smaller type. However, the type $\tau[\mu\alpha.\tau/\alpha]$ is not necessarily smaller than $\mu\alpha.\tau$. So we cannot use $\mathcal{V}[\![\tau[\mu\alpha.\tau/\alpha]]\!]$ to define $\mathcal{V}[\![\mu\alpha.\tau]\!]$.

So to define these sets recursively, we need to recur on something else if we want to include recursive types.

The second difference in this definition is in the case for function types. We want to allow function values to be either explicit lambda expressions or recursive functions. By defining the set in terms of the elimination form for function types instead of the introduction form, we can implicitly include $\mathbf{rec}\, x.v$ as a value in this set, as long as it behaves like a recursive function.

Consider what would happen if we tried to do it the other way around, and add an alternative for recursively defined values to our prior definition for lambda terms. We cannot do this definition either because our definition is not well-founded: in the first line we define $\mathcal{V}[\![\tau_1 \to \tau_2]\!]$ in terms of $\mathcal{V}[\![\tau_1 \to \tau_2]\!]$.

$$\begin{array}{rcl}\mathcal{V}[\![\tau_1 \to \tau_2]\!] & = & \{\ \mathbf{rec}\, x.v \mid v[\mathbf{rec}\, x.v/x] \in \mathcal{V}[\![\tau_1 \to \tau_2]\!]\ \} \cup \\ & & \{\ \lambda x.e \mid \forall v2,\ v_2 \in \mathcal{V}[\![\tau_1]\!]\ \textit{implies}\ e[v_2/x] \in \mathcal{C}[\![\tau_2]\!]\ \}\end{array}$$

6.1.1 Semantic lemmas with small-step evaluation

Continuing our (doomed) proof attempt, we can restate some of the definitions that were present in the previous proof. We need to know when a substitution only includes values in the semantic set, and can use this to define a semantic form of the value and computation typing relations.

Definition 6.1.2 (Semantic substitution and typing)**.**

1. Define $\sigma \in \mathcal{G}[\![\Gamma]\!]$ when $\forall x \in \mathsf{dom}\,\Gamma, \sigma\, x \in \mathcal{V}[\![\Gamma\, x]\!]$.
2. Define $\Gamma \vDash e : \tau$ when forall $\sigma \in \mathcal{G}[\![\Gamma]\!]$, $e[\sigma] \in \mathcal{C}[\![\tau]\!]$.
3. Define $\Gamma \vDash v : \tau$ when forall $\sigma \in \mathcal{G}[\![\Gamma]\!]$, $v[\sigma] \in \mathcal{V}[\![\tau]\!]$.

Because we are working with a small-step semantics, we will also need the following property.

Lemma 6.1.1 (Reverse evaluation)**.** If $e \rightsquigarrow e'$ and $e' \in \mathcal{C}[\![\tau]\!]$ then $e \in \mathcal{C}[\![\tau]\!]$.

Proof. The proof is by definition—we just extend the multi-step evaluation one more step. □

Now let's think about which of our semantic typing rules are true. Some go through very similarly to their analogs for the STLC (and the big-step semantics). And in some places the division between value typing rules and computation typing rules cleans things up nicely. No more need to reason about how values evaluate to themselves.

The base cases are straightforward.

Lemma 6.1.2 (Semantic var rule)**.** $\Gamma \vDash x : \Gamma\, x$.

Lemma 6.1.3 (Semantic lit rule)**.** $\Gamma \vDash k : \mathbf{Nat}$.

Let's look that the case for (non-recursive) functions.

Lemma 6.1.4 (Semantic abs rule)**.** If $\Gamma, x\!:\!\tau \vDash e : \tau_1 \to \tau_2$, then $\Gamma \vDash \lambda x.e : \tau_1 \to \tau_2$.

Proof. Let $\sigma \in \mathcal{G}[\![\Gamma]\!]$ be arbitrary. We want to prove that for any $v_1 \in \mathcal{V}[\![\tau_1]\!]$, we have $(\lambda x.e)[\sigma]\ v_1 \in \mathcal{C}[\![\tau_2]\!]$. Using our assumption, we know that $e[v_1/x, \sigma] \in \mathcal{C}[\![\tau_2]\!]$. Because $\mathcal{C}[\![\tau_2]\!]$ is closed under reverse evaluation, and $(\lambda x.e)[\sigma]\ v_1 \rightsquigarrow e[v_1/x, \sigma]$ we are done. □

Lemma 6.1.5 (Semantic app rule)**.** If $\Gamma \vDash v_1 : \tau_1 \to \tau_2$ and $\Gamma \vDash v_2 : \tau_1$, then $\Gamma \vDash v_1\ v_2 : \tau_2$.

Proof. Let $\sigma \in \mathcal{G}[\![\Gamma]\!]$ be arbitrary. We want to prove that for any $v1v2[\sigma] \in \mathcal{C}[\![\tau_2]\!]$. By our assumptions, we know that $v_1[\sigma] \in \mathcal{V}[\![\tau_1 \to \tau_2]\!]$ and $v_2[\sigma] \in \mathcal{V}[\![\tau_1]\!]$. By definition, we know our desired result. □

We have isolated all of our computation to let expressions.

Lemma 6.1.6 (Semantic let rule)**.** If $\Gamma \vDash e_1 : \tau_1$ and $\Gamma, x : \tau_1 \vDash e_2 : \tau_2$, then $\Gamma \vDash$ $\mathbf{let}\ x\ = e_1\ \mathbf{in}\ e_2 : \tau_2$.

Proof. Let $\sigma \in \mathcal{G}[\![\Gamma]\!]$ be arbitrary. We want to prove that for any $(\mathbf{let}\ x\ = e_1\ \mathbf{in}\ e_2)[\sigma] \in \mathcal{C}[\![\tau_2]\!]$. By our assumptions we know that $e_1[\sigma] \in \mathcal{C}[\![\tau_1]\!]$. This means that there is some $e_1[\sigma] \rightsquigarrow^* v_1$ and $v_1 \in \mathcal{V}[\![\tau_1]\!]$. Furthermore by assumption, we know that $e_2[v_1/x, \sigma] \in \mathcal{C}[\![\tau_2]\!]$. We can put these evaluation sequences together to have $\mathbf{let}\ x\ = e_1\ \mathbf{in}\ e_2 \rightsquigarrow^* \mathbf{let}\ x\ = \mathbf{ret}\ v_1\ \mathbf{in}\ e_2 \rightsquigarrow e_2[v_1/x, \sigma]$. And, because the computation set is closed under reverse evaluation, we are done. □

6.1.2 A problem

Our typing rule for recursive values looks like this.

Lemma 6.1.7 (Semantic rec rule)**.** If $\Gamma, x\!:\!\tau \vDash v : \tau$ then $\Gamma \vDash \mathbf{rec}\,x.v : \tau$.

Let's consider the case only when τ is a function type, i.e. $\tau = \tau_1 \to \tau_2$.

Let $\sigma \in \mathcal{G}[\![\Gamma]\!]$ be arbitrary. We want to show that $\mathbf{rec}\,x.v[\sigma] \in \mathcal{V}[\![(\tau_1 \to \tau_2)]\!]$. Let $v_1 \in \mathcal{V}[\![\tau_1]\!]$ be arbitrary. We now want to show that

$$\mathbf{rec}\,x.v[\sigma]\ v_1 \in \mathcal{C}[\![\tau_2]\!]$$

By closure under reverse evaluation, this is the same as showing that

$$v[\mathbf{rec}\,x.v[\sigma]/x, \sigma]\ v_1 \in \mathcal{C}[\![\tau_2]\!]$$

By assumption, we can conclude this as long as we have

$$\mathbf{rec}\,x.v[\sigma]/x, \sigma \in \mathcal{G}[\![\Gamma, x\!:\!\tau_1 \to \tau_2]\!]$$

This requires showing that

$$\mathbf{rec}\,x.v[\sigma] \in \mathcal{V}[\![\tau_1 \to \tau_2]\!]$$

But now we are stuck! This is *exactly* the result that we were already trying to show.

6.2 Fixing the problem: counting evaluation steps

In the previous section, we saw that there were two problems with the approach. The first is that if we want to include recursive types, we cannot define the value relation by recursion over type structure. The semantics of recursive types are defined in terms of themselves, not in terms of smaller components.

The second issue is that when we are trying to prove that recursive values are in the relation, because they do a self-substitution, we need to know that they are already in the relation to show that they are in the relation.

Now, if our semantic sets were coinductively defined, we would solve both problems. We could define the set in terms of itself (allowing recursive types) and to show that an element was a member of a set, we could use coinduction. But, we cannot do that with a straightforward definition. The case for function types features a negative occurrence of the value set, so the naive definition cannot be used in a coinductive relation.

Instead, we are going to define these sets a little less directly.

Recall this definition of "runs safely" from before:

Definition 6.2.1 (Safe for k)**.** A program e runs safely for k steps in two ways. If e cannot step, then e must be some $\mathbf{ret}\ v$. Otherwise, if it steps $e \rightsquigarrow e'$ and $0\langle k$ then e' must run safely for $k-1$ steps.

Definition 6.2.2 (Runs safely)**.** A program runs safely if for all k, it is safe for k.

The definition of safe for k is inductively defined over k. It holds for all programs that do not get stuck within k steps. By requiring that this predicate hold for all k, we can define type safety without requiring programs to terminate. Diverging programs satisfy this relation because they do not get stuck after any number of steps of evaluation.

Lemma 6.2.1 (Safe programs don't get stuck)**.** If e is safe, then for all $e \rightsquigarrow^* e'$, either e' is $\mathbf{ret}\ v$, or e' can take another step.

Another way to say this is to think about sets.

Let S_k be the set of all programs that are safe for k steps. That means that we have an infinite chain of decreasing sets.

$$S_0 \subseteq S_1 \supseteq S_2 \supseteq S_3 \ldots$$

This sequence is *downward closed*: for each $i \leq j$ we have $S_j \subseteq S_i$.

Furthermore, the set of safe programs is the intersection of all of the sets in this sequence.

$$S = \bigcap_i S_i$$

What we have just done is define a set by coinduction. Another way to set up this definition of safety is to consider this function from sets of terms to sets of terms.

$$FX = \{ e\ \mathbf{irreducible}\ \textit{implies} \quad e = \mathbf{ret}\ v \text{ or } e \rightsquigarrow e' \text{ implies } e' \in X\ \}$$

Say $F^i X$ is i applications of F to X. Then we have:

$$S = \bigcap_i F^i \mathcal{U}$$

The function F is monotonic: $X \subseteq Y$ implies that $FX \subseteq FY$, so we can prove a coinduction principle for this definition. This principle states that if we want to prove that some element is in S, then we need to show that it is in some set S_1, such that $S_1 \subseteq F(S_1)$.

For example, say we have some term $(\mathbf{rec}\, f.\lambda x.f\ x)\ 3$ and we want to show that this term runs safely. Intuitively we know that this is the case because we have the evaluation sequence:

$$(\mathbf{rec}\, f.\lambda x.f\ x)\ 3 \rightsquigarrow (\lambda x.(\mathbf{rec}\, f.\lambda x.f\ x))\ 3 \rightsquigarrow (\mathbf{rec}\, f.\lambda x.f\ x)\ 3 \rightsquigarrow \ldots$$

Then we can pick S_1 to be the set containing just this term and its unfolding. Our proof obligation is to show that $S_1 \in FS_1$, i.e. that both terms step to an element of S_1.

6.3 Step-indexed semantic safety

Let's use this idea to define our logical relation as a downward-closed step-indexed sequence of sets.

As before, we'll split our logical relation into two parts, mutually defining sets of values and sets of terms (computations). What is different this time is that these sets are also indexed by a step-count. So our relations are now three place relations, between types, values or terms, and step counts. This time, we write $\mathcal{V}[\![\, v \in \tau \,]\!]_k$ and $\mathcal{C}[\![\, e \in \tau \,]\!]_k$, when v and e are in their respective relations at step index k.

Furthermore, our computation set will contain terms that don't get stuck.

Lemma 6.3.1 (Semantic safety)**.** If for any k, we have $\mathcal{C}[\![\, e \in \tau \,]\!]_k$, then for all $e \rightsquigarrow^* e'$, either e' is $\mathbf{ret}\ v$, or e' can take another step.

Our fundamental lemma will be to show that closed, well-typed terms $\vdash e \in \tau$ are in $\mathcal{C}[\![\, e \in \tau \,]\!]_k$ for any k. This will give us (the hard way) a type safety theorem for statically well-typed terms.

While this result is a bit underwhelming (the preservation/progress based theorem is a lot easier) it will give us a framework that we can extend to prove more interesting theorems.

6.3.1 Step-indexed propositions

Sometimes, we want to work with our relations as functions from steps to propositions. For example, if we write $\mathcal{V}[\![\, v \in \tau \,]\!]$, without the argument k, we are talking about a function. There are two meta-operations on step-indexed propositions that we can use to clarify our reasoning.

The first operation is called the *later modality*, and takes a step-indexed proposition and produces a new step-indexed proposition. This operation accesses its argument at the previous step count. If the step is zero, then it holds trivially.

Definition 6.3.1 (Later modality)**.** Define $\triangleright_k \phi$ by case analysis on k.

- $\triangleright_0 \phi$ always holds

- $\triangleright_{(\mathbf{S}\,k)}\ \phi$ holds iff ϕ_k.

The next operation is called *stepped implication*, and takes *two* step-indexed propositions to produce a new step-indexed proposition.

Definition 6.3.2 (Step-indexed implication)**.** Define $\phi \Longrightarrow_k \phi'$ as

$$\textit{for all } j < k,\ \phi_j \textit{ implies } \phi'_j$$

Definition 6.3.3 (Downward closed)**.** A step-indexed proposition is *downward closed* when $j <= k$ we have ϕ_k *implies* ϕ_j.

Suppose ϕ and ϕ' are both downward closed. The step-indexed proposition, ϕ_k *implies* ϕ'_k is not downward closed. Assume $j \leq k$, and that ϕ_k *implies* ϕ'_k. We want to show that ϕ_j *implies* ϕ'_j. So assume ϕ_j. We want to show ϕ'_j. If we could show ϕ'_k, we would be done because ϕ' is downward closed. Furthermore, by assumption, it also suffices to show ϕ_k. In other words, we need ϕ_j *implies* ϕ_k. But here we are stuck! We cannot use the fact that ϕ is downward closed here because that gives us the opposition relation between ϕ_j and ϕ_k.

Lemma 6.3.2 (Step-indexed implication is downward closed)**.** For any ϕ and ϕ', $\phi \Longrightarrow \phi'$ is downward closed.

Proof. Assume that $j <= k$ and $\phi \Longrightarrow_k \phi'$. We want to show that $\phi \Longrightarrow_j \phi'$. Unfolding definitions, assume we have some $i\langle j$ and we want to show that ϕ_i *implies* ϕ'_i. But, we are done, because by transitivity, we have $i\langle k$, so we can use our original assumption. □

6.3.2 Logical relation

Now let's define our the logical relation by *well-founded recursion* on the step-count k.

$$\begin{array}{lcl}
\mathcal{C}[\![\, e \in \tau \,]\!]_k & = & e\ \mathbf{irreducible}\ \textit{implies that there exists } v \textit{ such that} \\
 & & \qquad\qquad e = \mathbf{ret}\ v \textit{ and } \mathcal{V}[\![\, v \in \tau \,]\!]_k \\
 & & \textit{and } e \rightsquigarrow e' \textit{ implies } \triangleright_k\ \mathcal{C}[\![\, e' \in \tau \,]\!] \\
 & & \\
\mathcal{V}[\![\, v \in \mathbf{Void} \,]\!]_k & = & \textit{never} \\
\mathcal{V}[\![\, v \in \mathbf{Nat} \,]\!]_k & = & v \in \mathbb{N} \\
\mathcal{V}[\![\, \lambda x.e \in \tau_1 \to \tau_2 \,]\!]_k & = & \forall v_1, \mathcal{V}[\![\, v_1 \in \tau_1 \,]\!] \Longrightarrow_k \mathcal{C}[\![\, e[v_2/x] \in \tau_2 \,]\!] \\
\mathcal{V}[\![\, \mathbf{rec}\, x.v \in \tau_1 \to \tau_2 \,]\!]_k & = & \triangleright_k\ \mathcal{V}[\![\, v[\mathbf{rec}\, x.v/x] \in \tau_1 \to \tau_2 \,]\!] \\
\mathcal{V}[\![\, (v_1, v_2) \in \tau_1 * \tau_2 \,]\!]_k & = & \triangleright_k\ \mathcal{V}[\![\, v_1 \in \tau_1 \,]\!] \textit{ and } \triangleright_k\ \mathcal{V}[\![\, v_3 \in \tau_2 \,]\!] \\
\mathcal{V}[\![\, \mathbf{rec}\, x.v \in \tau_1 * \tau_2 \,]\!]_k & = & \triangleright_k\ \mathcal{V}[\![\, v[\mathbf{rec}\, x.v/x] \in \tau_1 * \tau_2 \,]\!] \\
\mathcal{V}[\![\, \mathbf{inj}_1 v_1 \in \tau_1 + \tau_2 \,]\!]_k & = & \triangleright_k\ \mathcal{V}[\![\, v_1 \in \tau_1 \,]\!] \\
\mathcal{V}[\![\, \mathbf{inj}_2 v_2 \in \tau_1 + \tau_2 \,]\!]_k & = & \triangleright_k\ \mathcal{V}[\![\, v_2 \in \tau_2 \,]\!] \\
\mathcal{V}[\![\, \mathbf{fold}\, v \in \mu\alpha.\tau \,]\!]_k & = & \triangleright_k\ \mathcal{V}[\![\, v \in \tau[\mu\alpha.\tau/\alpha] \,]\!]
\end{array}$$

This definition is well-defined because recursive occurrences of $\mathcal{C}[\![\, e \in \tau \,]\!]$ and $\mathcal{V}[\![\, v \in \tau \,]\!]$ are done with steps that are strictly smaller than k. (There is one exception: definition of $\mathcal{C}[\![\, e \in \tau \,]\!]_k$ refers to $\mathcal{V}[\![\, v \in \tau \,]\!]_k$ at the same index. However,

$\mathcal{V}[\![\, v \in \tau \,]\!]_k$ makes all of its recursive calls to smaller indices so we never get back to $\mathcal{C}[\![\, e \in \tau \,]\!]_k$.)

The way this works is through the use of the two auxiliary operations defined above: if ϕ is a step-indexed set, then $\triangleright\phi_k$ accesses ϕ at the predecessor of k. The use of this operation in $\mathcal{C}[\![\, e \in \tau \,]\!]_k$ means that after taking a step, the next term must be in some smaller set.

For the case of product types, we include all pairs, where both components are in the previous set, and all recursive values, where the unrolling is in the previous set.

We do a similar thing for functions, including explicitly lambda terms and recursive values. However, the lambda case uses the step-indexed implication defined above. This makes sure that we will be able to prove that $\mathcal{C}[\![\, e \in \tau \,]\!]$ and $\mathcal{V}[\![\, v \in \tau \,]\!]$ are downward closed.

Lemma 6.3.3 (Downward closed)**.**

1. If $j \leq k$ then $\mathcal{V}[\![\, v \in \tau \,]\!]_k$ *implies* $\mathcal{V}[\![\, v \in \tau \,]\!]_j$.
2. If $j \leq k$ then $\mathcal{C}[\![\, e \in \tau \,]\!]_k$ *implies* $\mathcal{C}[\![\, e \in \tau \,]\!]_j$.

6.3.3 Semantic typing and semantic typing lemmas

Continuing on, we find that our semantic substitution and semantic typing propositions are also now step-indexed. Furthermore, to make sure that the typing relations are downward closed, we use step-indexed implication.

Definition 6.3.4 (Semantic substitution and typing)**.**

1. Define $[\![\sigma \in \Gamma]\!]_k$ when for all $x \in \mathsf{dom}\,\Gamma$, we have $\mathcal{V}[\![\, \sigma\, x \in \Gamma\, x \,]\!]_k$.
2. Define $\Gamma \vDash_k e \in \tau$ when $[\![\sigma \in \Gamma]\!] \Longrightarrow_k \mathcal{C}[\![\, e[\sigma] \in \tau \,]\!]$.
3. Define $\Gamma \vDash_k v \in \tau$ when $[\![\sigma \in \Gamma]\!] \Longrightarrow_k \mathcal{V}[\![\, v[\sigma] \in \tau \,]\!]$.

Lemma 6.3.4 (Semantic var rule)**.** forall k, we have $\Gamma \vDash_k x \in \Gamma\, x$.

Proof. We assume some k, $j \langle k$, and $[\![\sigma \in \Gamma]\!]_j$. We want to show $\mathcal{V}[\![\, x[\sigma] \in (\Gamma\, x) \,]\!]_j$. But this is true by unfolding definitions. ☐

Lemma 6.3.5 (Semantic lit rule)**.** forall k, $\Gamma \vDash_k i \in \mathbf{Nat}$.

We assume some k, $j \langle k$, and $[\![\sigma \in \Gamma]\!]_j$. We want to show $\mathcal{V}[\![\, i[\sigma] \in \mathbf{Nat} \,]\!]_j$. But this is again true by unfolding definitions.

Lemma 6.3.6 (Semantic abs rule)**.** forall k, if $\Gamma, x : \tau \vDash_k e \in \tau_1 \to \tau_2$, then $\Gamma \vDash_k \lambda x.e \in \tau_1 \to \tau_2$.

Proof. We assume some k, $j \langle k$, and $[\![\sigma \in \Gamma]\!]_j$. We want to prove that $\mathcal{V}[\![\, (\lambda x.e)[\sigma] \in \tau_1 \to \tau_2 \,]\!]_j$, which can be restated as for any $i \langle j$, $\mathcal{V}[\![\, v \in \tau_1 \,]\!]_i$ implies $\mathcal{C}[\![\, e[v/x, \sigma] \in \tau_2 \,]\!]_i$. Using our assumption, we know that because $i < k$, if $[\![v/x, \sigma \in \Gamma, x : \tau_1]\!]_i$ then $\mathcal{C}[\![\, e[v/x, \sigma] \in \tau_2 \,]\!]_i$. So it suffices to show that $\mathcal{V}[\![\, v \in \tau_1 \,]\!]_i$ and $[\![\sigma \in \Gamma]\!]_i$. However, we assumed the former, and the latter holds because $[\![\sigma \in \Gamma]\!]$ is downward closed. ☐

Before we continue further, what if our definition for function types had instead been in terms of application, instead of substitution. i.e. we combined both cases above into the single case:

$$\mathcal{V}[\![v \in \tau_1 \to \tau_2]\!]_k = \forall v_1, \mathcal{V}[\![v_1 \in \tau_1]\!] \Longrightarrow_k \mathcal{C}[\![v\ v_2 \in \tau_2]\!]$$

We assume some k, $j\langle k$, and $[\![\sigma \in \Gamma]\!]_j$. We want to prove that $\mathcal{V}[\![(\lambda x.e)[\sigma] \in \tau_1 \to \tau_2]\!]_j$, which can be restated as for any $i\langle j$, $\mathcal{V}[\![v_1 \in \tau_1]\!]_i$ implies $\mathcal{C}[\![(\lambda x.e)[\sigma]\ v_1 \in \tau_2]\!]_i$. Using our assumption, we know that for any $i' < k$, if $[\![v/x, \sigma \in \Gamma, x : \tau_1]\!]_{i'}$ then $\mathcal{C}[\![e[v/x, \sigma] \in \tau_2]\!]_{i'}$. Our logical relation is backwards closed, but it takes a step. We need $\mathcal{C}[\![e[v/x, \sigma] \in \tau_2]\!]_{(\mathbf{S}\, i)}$ to show $\mathcal{C}[\![(\lambda x.e)[\sigma]\ v_1 \in \tau_2]\!]$. By instantiating i' with $\mathbf{S}\, i$, it suffices to show that $\mathcal{V}[\![v \in \tau_1]\!]_{(\mathbf{S}\, i)}$ and $[\![\sigma \in \Gamma]\!]_{(\mathbf{S}\, i)}$. However, here we are stuck. We know these results about i but not i'.

Now let's do the case where we were stuck before.

Lemma 6.3.7 (Semantic rec rule)**.** For all k, if $\Gamma, x : \tau \vDash_k v \in \tau$ then $\Gamma \vDash_k \mathbf{rec}\, x.v \in \tau$.

Proof. Let's again consider the case only when τ is a function type, i.e. $\tau = \tau_1 \to \tau_2$. We will prove this by strong induction on k. We can assume that the lemma holds for all indices less than k, and we want to show it for k.

Now, assume some $j\langle k$, and $[\![\sigma \in \Gamma]\!]_j$. We want to prove that $\mathcal{V}[\![(\mathbf{rec}\, x.v)[\sigma] \in \tau_1 \to \tau_2]\!]_j$, which can be restated as $\triangleright_j\ \mathcal{V}[\![v[\mathbf{rec}\, x.v/x, \sigma] \in \tau_1 \to \tau_2]\!]$. We can also assume that $j = Si$, as this is trivial otherwise. So we want to show $\mathcal{V}[\![v[\mathbf{rec}\, x.v/x, \sigma] \in \tau_1 \to \tau_2]\!]_i$. This result holds by instantiating our assumption with i and the substitution $\mathbf{rec}\, x.v/x, \sigma$. We also need to show that $[\![\mathbf{rec}\, x.v/x, \sigma \in \Gamma, x : \tau_1 \to \tau_2]\!]_i$. This holds because $[\![\sigma \in \Gamma]\!]$ is downward closed. So we really only need to show $\Gamma \vDash_i \mathbf{rec}\, x.v \in \tau$. But as $i < k$ we can use our induction hypothesis, and reduce this to $\Gamma, x : \tau \vDash_i v \in \tau$. However, semantic typing is also downward closed, and we assumed this judgment at k, so we can also have it at i. ☐

Compared to where we got stuck before, we needed the semantic typing of the recursive value, but only at a smaller index, which was available via induction.

Lemma 6.3.8 (Semantic app rule)**.** For all k, if $\Gamma \vDash_k v_1 \in \tau_1 \to \tau_2$ and $\Gamma \vDash_k v_2 \in \tau_1$, then $\Gamma \vDash k\ v_1\ v_2 : \tau_2$.

To prove this lemma, we can assume some $j\langle k$, and $[\![\sigma \in \Gamma]\!]_j$. That gives us a goal of $\mathcal{C}[\![v_1\ \ v_2 \in \tau_2]\!]_k$. We can prove this result with a sub-lemma, that applies after instantiating our premises with the assumptions.

Lemma 6.3.9 (Set app)**.** For all k, if $\mathcal{V}[\![v_1 \in \tau_1 \to \tau_2]\!]_k$ and $\mathcal{V}[\![v_2 \in \tau_1]\!]_k$, then $\mathcal{C}[\![v_1\ v_2 \in \tau_2]\!]_k$.

Proof. We will prove this by strong induction on k.

We want to prove that $\mathcal{C}[\![v_1\ v_2 \in \tau_2]\!]_k$. We have two cases for $\mathcal{V}[\![v_1 \in \tau_1 \to \tau_2]\!]_k$:

1. If v_1 is $\mathbf{rec}\, x.v$, then we know that $\triangleright_k\ \mathcal{V}[\![v[\mathbf{rec}\, x.v/x] \in \tau_1 \to \tau_2]\!]$. We want to show two things: If e **irreducible** then there exists some v such that $e = \mathbf{ret}\, v$ *and* $\mathcal{V}[\![v \in \tau]\!]_k$ and if $e \rightsquigarrow e'$ then $\triangleright_k\ \mathcal{C}[\![e' \in \tau_2]\!]$. The first case does not apply because $(\mathbf{rec}\, x.v)\ v_2$ is always reducible, as it steps to $v[\mathbf{rec}\, x.v/x]\ \ v_2$. So we focus on the second, where we need to show that $\triangleright_k\ \mathcal{C}[\![v[\mathbf{rec}\, x.v/x]\ v_2 \in \tau_2]\!]$. Here we can assume that $k = Sj$ as the zero case is trivial. So we need to show that $\mathcal{C}[\![v[\mathbf{rec}\, x.v/x]\ v_2 \in \tau_2]\!]_j$. As j is less than k, we can use our induction hypothesis, with our initial assumption about $v1$ and the downward closure of $\mathcal{C}[\![v_2 \in \tau_2]\!]$.

2. If v_1 is $\lambda x.e$, then we know that $\forall v_1, \mathcal{V}[\![\, v_2 \in \tau_1 \,]\!] \Longrightarrow_k \mathcal{C}[\![\, e[v_2/x] \in \tau_2 \,]\!]$. Again we want to show two things: If e **irreducible** then there exists some v such that $e = \mathbf{ret}\ v$ *and* $\mathcal{V}[\![\, v \in \tau \,]\!]_k$ and if $e \rightsquigarrow e'$ then $\triangleright_k \mathcal{C}[\![\, e' \in \tau_2 \,]\!]$. And again we have $(\lambda x.e)\ v_2 \rightsquigarrow e[v_2/x]$ so the first case doesn't apply. So we need to show $\triangleright_k \mathcal{C}[\![\, e[v_2/x] \in \tau_2 \,]\!]$. Again we assume that $k = Sj$, and reduce this to $\mathcal{C}[\![\, e[v_2/x] \in \tau_2 \,]\!]_j$. By our assumption about v_1, as j is strictly smaller than k, we only need to show $\mathcal{V}[\![\, v_2 \in \tau_1 \,]\!]_j$. This holds by downward closure.

□

Now let's prove the rules specific to fine-grained CBV. In particular, we have isolated all sequencing to **let** terms, so here is where we will need to consider how our computation set is closed under evaluation.

Lemma 6.3.10 (Semantic let rule)**.** For all k, if $\Gamma \vDash_k e_1 \in \tau_1$ and $\Gamma, x : \tau_1 \vDash_k e_2 \in \tau_2$, then $\Gamma \vDash_k \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \in \tau_2$.

Like the application rule, we will show this result with the help of a sub-lemma.

Lemma 6.3.11. For all k, if $\mathcal{C}[\![\, e_1 \in \tau_1 \,]\!]_k$ and $\forall v, \mathcal{V}[\![\, v \in \tau_1 \,]\!] \Longrightarrow_k \mathcal{C}[\![\, \mathbf{let}\ x = \mathbf{ret}\ v\ \mathbf{in}\ e_2 \in \tau_2 \,]\!]$ then $\mathcal{C}[\![\, (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) \in \tau_2 \,]\!]_k$.

6.4 Further Reading

Step-indexed logical relations were invented by Appel and McAllester [AM01] and developed by Ahmed [Ahm04].

This methodology forms the foundation of the Iris logic. Timany et al [TKDB24] describes how to prove this and more sophisticated results using Iris, while hiding the step-counts altogether using a special purpose logic.

The approach taken in this section was inspired by Xavier Leroy's lectures on step-indexed logical relations, from the lecture series titled "Programming = proving? The Curry-Howard correspondence today". Slides available from `https://xavierleroy.org/CdF/2018-2019/8.pdf`.

Additional reading about the foundations of inductive and coinductive definitions can be found in Ron Garcia's lecture notes available from `https://www.cs.ubc.ca/~rxg/cpsc509-spring-2022/06-coinduction.pdf`

Bibliography

[Ahm04] Amal Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.

[AM01] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, September 2001.

[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.

[Cha13] Arthur Charguéraud. Pretty-big-step semantics. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 41–60, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[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.

[FSSS19] Yannick Forster, Steven Schäfer, Simon Spies, and Kathrin Stark. Call-by-push-value in coq: operational, equational, and denotational theory. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, page 118–131, New York, NY, USA, 2019. Association for Computing Machinery.

[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.

[Kah87] G. Kahn. Natural semantics. In Franz J. Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS 87*, pages 22–39, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[Ler06] Xavier Leroy. Coinductive big-step operational semantics. In Peter Sestoft, editor, *Programming Languages and Systems*, pages 54–68, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[LPT03] PaulBlain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Information and Computation*, 185(2):182–210, 2003.

[Men87] Paul Francis Mendler. Inductive definitions in type theory, 1987.

[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.

[SNO+07] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: effective tool support for the working semanticist. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, page 1–12, New York, NY, USA, 2007. Association for Computing Machinery.

[TKDB24] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. A logical approach to type soundness. *J. ACM*, 71(6), November 2024.

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