Programming Languages: Semantics and Types

Stephanie Weirich

October 8, 2025

Contents

Preface **v**

What is this course all about? **vii**

1 Type and Effect systems **1**
- 1.1 Core system 1
  - 1.1.1 Generalizing effects 3
  - 1.1.2 Recursive values and types 4
- 1.2 Syntactic metatheory: Preservation and progress 5
- 1.3 Effect soundness 5

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 and Effect systems

The REC language introduced the idea of a computational effect: *nontermination*.

However, this effect is invisible to the type system. As a result, we had to take a pessimistic view of REC terms: any of them could diverge at any time.

In this chapter, we will refine our type system so that its static analysis can say more than just the form of the resulting value (if any). In particular, we will annotate our typing judgent with an effect modality ε that describes the potential *effects* of computation (if any). For non-termination, effects can either be $\bot$ (indicating that we know the code terminates) or DIV, indicating that it may diverge. However, the structure of the type-and-effect system that we present is more general than that, and could be extended to track other forms of effects.

1.1 Core system

For concreteness, we continue to work with fine-grained CBV REC language, reusing its syntax of terms and values and its small-step operational semantics. The key updates are all in the type system: we slightly modify the syntax of types and we present a completely new pair of typing judgements.

$$\begin{array}{llll} \text{effect flags} & \varepsilon & ::= & \bot \mid \mathsf{DIV} \\ \\ \text{types} & \tau & ::= & \mathbf{Void} \mid \mathbf{Nat} \mid \tau_1 \xrightarrow{\varepsilon} \tau_2 \mid \tau_1 \overset{\varepsilon}{\times} \tau_2 \mid \tau_1 + \tau_2 \end{array}$$

The necessary update to the type syntax is to annotate the types of values that can be recursive (i.e. function types and product types) with an effect flag ε. A recursively-defined function or product **must** use DIV in its type to indicate that they may diverge. Non-recursive products **can** use $\bot$, indicating that they are safe for termination. However some non-recursive functions must use DIV if they could diverge when applied, for example, if their body contains an infinite loop. We say that the flags on this type describe the *latent* effect of the function: effects that don't happen when the function is created but could occur when the function is used.

As in the previous section, we start with the typing rules for the core language, deferrring recursive values and recursive types to the next subsection. These rules are an example of a *type-and-effect* system and they are, in fact, general about the specific effects that are tracked by the type system.

Definition 1.1.1 (Type system)**.**

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

$$\frac{}{\Gamma \vdash 0 \in \mathbf{Nat}}\ \textsc{tv-zero} \qquad \frac{\Gamma \vdash v \in \mathbf{Nat}}{\Gamma \vdash \mathbf{S}\, v \in \mathbf{Nat}}\ \textsc{tv-succ} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x \in \tau}\ \textsc{tv-var} \qquad \frac{\Gamma, x{:}\tau_1 \vdash e \overset{\varepsilon}{\in} \tau_2}{\Gamma \vdash \lambda x.e \in \tau_1 \xrightarrow{\varepsilon} \tau_2}\ \textsc{tv-abs-eff}$$

$$\frac{\Gamma \vdash v_1 \in \tau_1 \qquad \Gamma \vdash v_2 \in \tau_2}{\Gamma \vdash (v_1, v_2) \in \tau_1 \overset{\varepsilon}{\times} \tau_2}\ \textsc{tv-pair-eff} \qquad \frac{\Gamma \vdash v_1 \in \tau_1}{\Gamma \vdash \mathbf{inj}_1 v_1 \in \tau_1 + \tau_2}\ \textsc{tv-inj1} \qquad \frac{\Gamma \vdash v_2 \in \tau_2}{\Gamma \vdash \mathbf{inj}_2 v_2 \in \tau_1 + \tau_2}\ \textsc{tv-inj2}$$

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

$$\frac{\Gamma \vdash v \in \tau}{\Gamma \vdash \mathbf{ret}\, v \overset{\bot}{\in} \tau}\ \textsc{tee-ret} \qquad \frac{\Gamma \vdash e_1 \overset{\varepsilon_1}{\in} \tau_1 \qquad \Gamma, x{:}\tau_1 \vdash e_2 \overset{\varepsilon_2}{\in} \tau}{\Gamma \vdash \mathbf{let}\ x\ = e_1\ \mathbf{in}\ e_2 \overset{\varepsilon_1 \oplus \varepsilon_2}{\in} \tau}\ \textsc{tee-let} \qquad \frac{\Gamma \vdash v_1 \in \tau_1 \xrightarrow{\varepsilon} \tau_2 \qquad \Gamma \vdash v_2 \in \tau_1}{\Gamma \vdash v_1\ v_2 \overset{\varepsilon}{\in} \tau_2}\ \textsc{tee-app}$$

$$\frac{\Gamma \vdash v \in \mathbf{Nat} \qquad \Gamma \vdash e_1 \overset{\varepsilon}{\in} \tau \qquad \Gamma, x_2{:}\mathbf{Nat} \vdash e_2 \overset{\varepsilon}{\in} \tau}{\Gamma \vdash \mathbf{case}\, v\, \mathbf{of}\ \{0 \Rightarrow e_1;\ \mathbf{S}\, x_2 \Rightarrow e_2\} \overset{\varepsilon}{\in} \tau}\ \textsc{tee-ifz} \qquad \frac{\Gamma \vdash v \in \tau_1 \overset{\varepsilon}{\times} \tau_2}{\Gamma \vdash \mathbf{prj}_1 v \overset{\varepsilon}{\in} \tau_1}\ \textsc{tee-prj1} \qquad \frac{\Gamma \vdash v \in \tau_1 \overset{\varepsilon}{\times} \tau_2}{\Gamma \vdash \mathbf{prj}_2 v \overset{\varepsilon}{\in} \tau_2}\ \textsc{tee-prj2}$$

$$\frac{\Gamma \vdash v \in \tau_1 + \tau_2 \qquad \Gamma, x_1{:}\tau_1 \vdash e_1 \overset{\varepsilon}{\in} \tau \qquad \Gamma, x_2{:}\tau_2 \vdash e_2 \overset{\varepsilon}{\in} \tau}{\Gamma \vdash \mathbf{case}\, v\, \mathbf{of}\ \{\mathbf{inj}_1 x_1 \Rightarrow e_1; \mathbf{inj}_2 x_2 \Rightarrow e_2\} \overset{\varepsilon}{\in} \tau}\ \textsc{tee-case} \qquad \frac{\Gamma \vdash e \overset{\varepsilon_1}{\in} \tau \qquad \varepsilon_1 <: \varepsilon_2}{\Gamma \vdash e \overset{\varepsilon_2}{\in} \tau}\ \textsc{tee-sub-eff}$$

Values do not have effects, so we do not need to change their typing rules, execpt for the introduction form for functions. In that case, we record the effect of the body of the function in the function type.

In the effect annotated computation rules, the judgment form $\Gamma \vdash e \overset{\varepsilon}{\in} \tau$ uses ε to indicate whether the computation e is pure (i.e. terminates) or may diverge. Returned values must be pure, so rule TEE-RET uses $\bot$ for this annotation. However, a let binding sequences the evaluation of two computations e_1 and e_2. Therefore, the effect of the whole computation can be computed from the effect of these two subterms, using the operation $\varepsilon_1 \oplus \varepsilon_2$. For nontermination, this computation returns DIV if either effect is DIV and bottom otherwise.

In the application rule, the effect of the computation is completely determined by the latent effect of the function. If the function could diverge, for example if it is $\lambda x.loop$ or $\lambda x.\mathbf{case}\ x\ \mathbf{of}\ \{0 \Rightarrow loop;\ \mathbf{S}\ y \Rightarrow y\}$, then its type will include the DIV effect. On the other hand, if the function terminates on *all* arguments, then the type uses the $\bot$ effect. If the function is higher-order, i.e. if it takes another function as an argument, then the effect on the type of that argument determines the effect on the whole function. For example, we can give the function $\lambda f.f\ (f\ 3)$ the type $(\mathbf{Nat} \overset{\bot}{\to} \mathbf{Nat}) \overset{\bot}{\to} \mathbf{Nat}$ or the type $(\mathbf{Nat} \overset{\mathsf{DIV}}{\to} \mathbf{Nat}) \overset{\mathsf{DIV}}{\to} \mathbf{Nat}$.

The two projections $\mathbf{prj}_1 v$ and $\mathbf{prj}_2 v$ use the effects from the product types. Because this language includes recursive values, it is possible for projection to cause divergence. For example, the computation

$$\mathbf{prj}_1(\mathbf{rec}\ x.(x, x)) \rightsquigarrow \mathbf{prj}_1(\mathbf{rec}\ x.(x, x), \mathbf{rec}\ x.(x, x)) \rightsquigarrow \ldots$$

does not terminate.

The two pattern-matching elimination forms require that the effects for both branches be the same (just as the types of the both branches must be the same). During type-checking time, if the scrutinee is a variable, we don't know which branch will execute. So the effects of the computation could either be the effects of the first branch or the effects of the second.

What if these two branches don't have the same effect, such as $\mathbf{case}\ x\ \mathbf{of}\ \{0 \Rightarrow loop;\ \mathbf{S}\ y \Rightarrow y\}$ from above. In this case, we include a *subsumption rule*,rule TEE-SUB-EFF which allows the type system to weaken its analysis. Every effect system has a (pre)-order on effects. For nontermination, we have $\bot <:$ DIV, meaning that if we know that a system always terminates, then it is sound to weaken that analysis to say that it might diverge.

With this subeffecting rule, this type system becomes our first example of a system that is not syntax directed. In every other system that we have encountered so far, we have had a 1-1 correspondence between typing rules and syntax forms. This gives us strong inversion properties: If we have a derivation, we know exactly what order the rules must be applied for the system to type check.[1]

1.1.1 Generalizing effects

This type system is specialized to tracking non-termination. However, we can view these rules more abstractly if we consider effects to be an abstract structure.

Looking at the rules, we need the following operations from our abstract structure:

ε	some type of effect annotation
$\bot$	annotation for "pure" code
$\varepsilon_1 \oplus \varepsilon_2$	combination of effects
$\varepsilon_1 <: \varepsilon_2$	ordering of effects

It will turn out, that to show that our type-and-effect system has the rules that we want, we will also want to assume some properties about this structure. In particular, we will want to make sure that this structure is a *pre-ordered monoid*.

Our structure is a monoid if it satisfies the following three properties:

1. Left identity: $\bot \oplus \varepsilon = \varepsilon$

2. Right identity: $\varepsilon \oplus \bot = \varepsilon$

3. Associativity: $(\varepsilon_1 \oplus \varepsilon_2) \oplus \varepsilon_3 = \varepsilon_1 \oplus (\varepsilon_2 \oplus \varepsilon_3)$

[1]Remember that even though a type system is syntax-directed, it does not need to have unique types. All of our type systems allow programs to be given multiple distinct types for some functions.

It is a preorder if $<:$ is reflexive and transitive. In other words, we have $\varepsilon <: \varepsilon$ and $\varepsilon_1 <: \varepsilon_2$ and $\varepsilon_2 <: \varepsilon_3$ implies $\varepsilon_1 <: \varepsilon_3$.

Finally, it is a preordered monoid if the ordering is compatible with the combining operation. In other words, we have $\varepsilon_1 <: \varepsilon_1'$ and $\varepsilon_2 <: \varepsilon_2'$ then $\varepsilon_1 \oplus \varepsilon_2 <: \varepsilon_1' \oplus \varepsilon_2'$.

It turns out that the specific structure that we use here, with the definition of ε as $\bot$ or DIV, the ordering $\bot <:$ DIV and the combining operation described above, satisfies these properties. This structure satisfies other properties as well, such as commutativity of $\varepsilon_1 \oplus \varepsilon_2$, but we will find that we won't need to use that property in our proofs.

To modify this type system to track other forms of effects, we need to modify the effect structure that we use accordingly, as long as it satisfies these laws.

1.1.2 Recursive values and types

The DIV effect is explicitly introduced to the type system through the typing rules for recursive values and types.

Definition 1.1.2 (Type system).

$\boxed{\tau \ \mathsf{ok}}$ *(a recursive value can have type τ)*

OK-FUN-EFF
$$\frac{}{(\tau_1 \overset{\mathsf{DIV}}{\to} \tau_2)\ \mathsf{ok}}$$

OK-PROD-EFF
$$\frac{}{(\tau_1 \overset{\mathsf{DIV}}{\times} \tau_2)\ \mathsf{ok}}$$

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

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

TV-FOLD
$$\frac{\Gamma \vdash v \in \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash \mathbf{fold}\, v \in \mu\alpha.\tau}$$

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

TEE-UNFOLD
$$\frac{\Gamma \vdash v \in \mu\alpha.\tau}{\Gamma \vdash \mathbf{unfold}\, v \overset{\mathsf{DIV}}{\in} \tau[\mu\alpha.\tau/\alpha]}$$

Recall that the introduction rule for recursive values has a precondition that states what types of values can be recursive. To make sure that the infinite loops that these values can create is accurately accounted for, we require function and product types to be marked with DIV when they are assigned to recursive definitions. This annotation is used when these values are eliminated: the application and project rules need to know whether the value could loop when it is in the active position.

Similarly, unfolding a value with a recursive type also triggers the DIV effect. (We pessimistically assume that all values with recursive types could cause nontermination, so there is no need to mark this in the recursive type itself.)

1.2 Syntactic metatheory: Preservation and progress

Because this type system is not syntax directed, we cannot easily use inversion. If we have a derivation $\Gamma \vdash e \overset{\varepsilon}{\in} \tau$, the rule SUB-EFF rule could always have been used, so it is difficult to say more about specific derivations.

Lemma 1.2.1 (Ret inversion). If $\Gamma \vdash v \overset{\varepsilon}{\in} \tau$ then $\Gamma \vdash v \in \tau$.

Lemma 1.2.2 (Let inversion). If $\Gamma \vdash \textbf{let}\ x\ = e_1\ \textbf{in}\ e_2 \overset{\varepsilon}{\in} \tau$ then there exists some ε_1 and ε_2 such that $\varepsilon_1 \oplus \varepsilon_2 <: \varepsilon$ and $\Gamma \vdash e_1 \overset{\varepsilon_1}{\in} \tau$ and $\Gamma \vdash e_2 \overset{\varepsilon_2}{\in} \tau$.

Lemma 1.2.3 (App inversion). If $\Gamma \vdash v_1\ v_2 \overset{\varepsilon}{\in} \tau_2$, then there exists some ε_1, such that $\Gamma \vdash v_1 \in \tau_1 \xrightarrow{\varepsilon_1} \tau_2$ and $\Gamma \vdash v_2 \in \tau_1$ and $\varepsilon_1 <: \varepsilon$.

Lemma 1.2.4 (Prj inversion). If $\Gamma \vdash prji\ v \overset{\varepsilon}{\in}\ taui$ then there exists some ε_1 such that $\varepsilon_1 <: \varepsilon$ and $\Gamma \vdash v \in \tau_1 \overset{\varepsilon_1}{\times} \tau_2$.

1.3 Effect soundness

To know whether the effect annotations in our type system are meaningful, we need to prove more than type safety — we need to argue that if a term has type $\vdash e \overset{\perp}{\in} \tau$ then it must terminate. As before, we can prove this result using a logical relation.

$$\begin{array}{lcl}
\mathcal{C}[\![\tau]\!]^{\mathsf{DIV}} & = & \{\ e \mid e \rightsquigarrow^* v\ \textit{and} \quad v \in \mathcal{V}[\![\tau]\!]\ \} \\
\mathcal{C}[\![\tau]\!]^{\perp} & = & \textit{all terms}
\end{array}$$

$$\begin{array}{lcl}
\mathcal{V}[\![\textbf{Void}]\!] & = & \emptyset \\
\mathcal{V}[\![\textbf{Nat}]\!] & = & \mathbb{N} \\
\mathcal{V}[\![\tau_1 \xrightarrow{\varepsilon} \tau_2]\!] & = & \{\ v \mid \forall v_1,\ v_1 \in \mathcal{V}[\![\tau_1]\!]\ \textit{implies}\ v\ v_1 \in \mathcal{C}[\![\tau_2]\!]^{\varepsilon}\ \} \\
\mathcal{V}[\![\tau_1 \overset{\varepsilon}{\times} \tau_2]\!] & = & \{\ v \mid \textbf{prj}_1 v \in \mathcal{C}[\![\tau_1]\!]^{\varepsilon}\ \textit{and}\ \textbf{prj}_2 v \in \mathcal{C}[\![\tau_2]\!]^{\varepsilon}\ \} \\
\mathcal{V}[\![\mu\alpha.\tau]\!] & = & \emptyset
\end{array}$$

Lemma 1.3.1 (Fundamental). If $\Gamma \vdash e \overset{\varepsilon}{\in} \tau$ then $\Gamma \vDash e : \tau @ \varepsilon$. If $\Gamma \vdash v \in \tau$ then $\Gamma \vDash v : \tau$.

Using the fundamental lemma above, and the definition of $\mathcal{C}[\![\tau]\!]^{\perp}$ we can show that the termination effect implies termination.

Lemma 1.3.2 (Effect soundness). If $\vdash e \overset{\perp}{\in} \tau$, then there exists some v, such that $e \rightsquigarrow^* v$.

Bibliography

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

[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.