Programming Language Semantics

Programming Language Design 2020

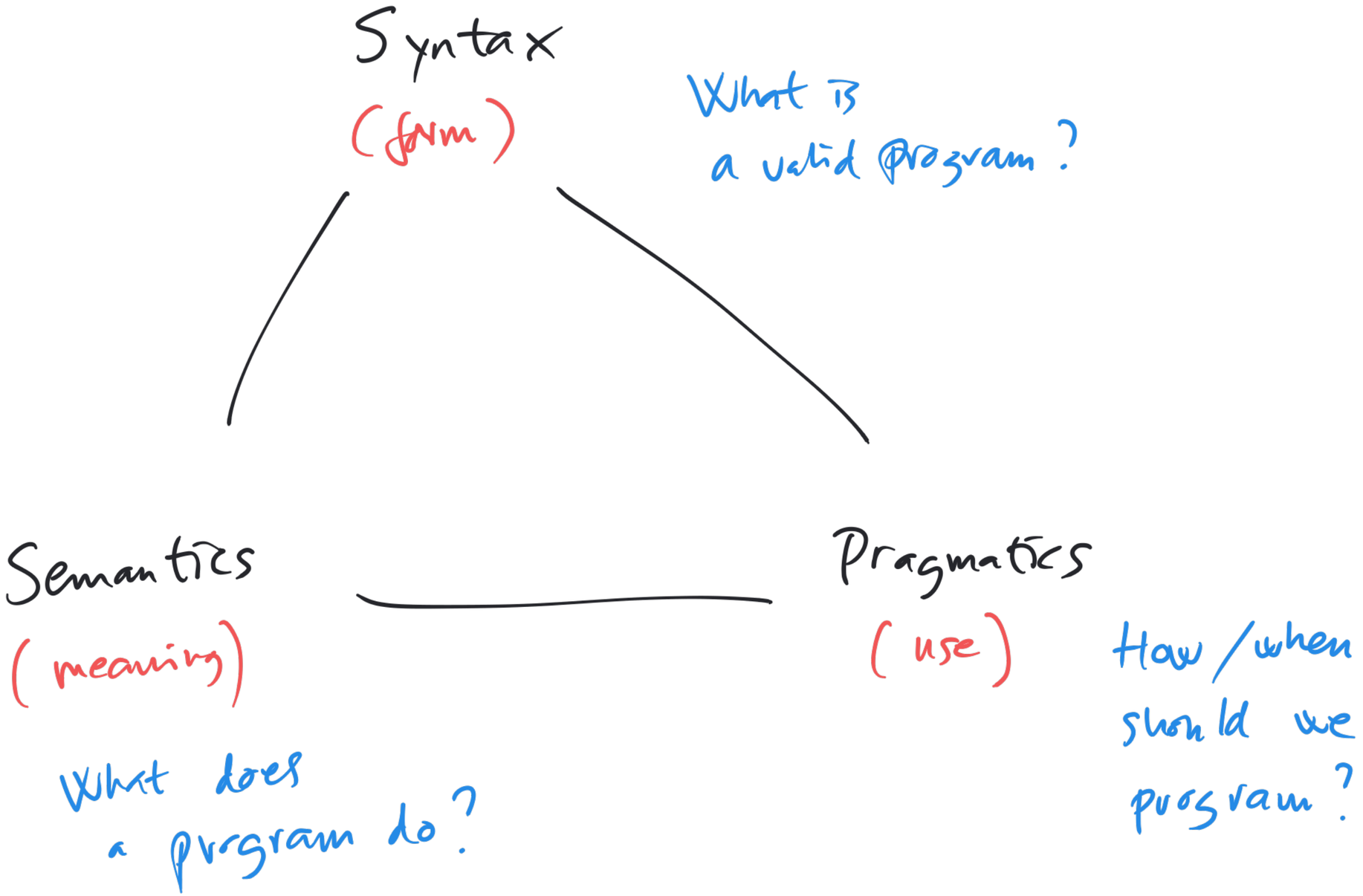
Hans Hüttel

15 March 2021

Part I

Motivation

Syntax and semantics (and pragmatics)



Semantics is a singular form!

The Greek word (transliterated) is *semantikós*, meaning 'significant'. The English word 'semantics' is a singular form, as are 'physics', 'mathematics' and other words that have similar Greek roots. So we speak of a semantics.

Why bother?

Can't we just refer to an implementation?

"The meaning of our program is what the compiler on some specific machine says it is!"

Can't we just refer to an implementation?

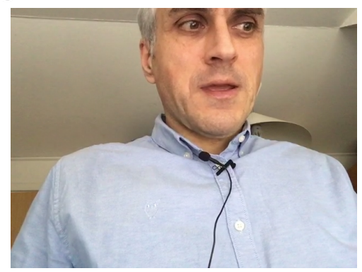
"The meaning of our program is what the compiler on some specific machine says it is!"

. . . but

- This is implementation-dependent!
- We cannot use such an approach for *program analysis* – all we could then analyse would be what the compiler produced.
- Where and how do we specify what the compiler should do? Everytime someone designs a new language, someone would have to write that first compiler.
- What do we do *when* the compiler becomes obsolete?

Can't we just use English?

The ALGOL 60 language was the result of very careful work by a committee of prominent researchers, including



- John Backus, the creator of FORTRAN (and originally a mathematician)
- John McCarthy, the creator of Lisp (and originally a mathematician)
- Peter Naur, the first Danish professor of computer science and founder of DIKU (and originally an astronomer).

ALGOL 60 was the first programming language whose syntax was defined formally. The notation used was a variant of context-free grammars, later known as Backus–Naur Normal Form (BNF). For the semantics of ALGOL 60 is concerned, the authors had to rely on very detailed descriptions in English.

The ALGOL 60 report

Reprinted from the COMMUNICATIONS OF THE ASSOCIATION FOR COMPUTING MACHINERY
Vol. 3, No. 5, May 1960
Made in U.S.A.
With typographical corrections as of June 28, 1960

Report on the Algorithmic Language ALGOL 60

PETER NAUR (*Editor*)

J. W. BACKUS, F. L. BAUER, J. GREEN, C. KATZ, J. MCCARTHY, A. J. PERLIS, H. RUTISHAUSER, K. SAMELSON, B. VAUQUOIS, J. H. WEGSTEIN, A. VAN WIJNGAARDEN, M. WOODGER

Dedicated to the Memory of WILLIAM TURANSKI

INTRODUCTION

Background

After the publication of a preliminary report on the algorithmic language ALGOL,[1, 2] as prepared at a conference in Zürich in 1958, much interest in the ALGOL language developed.

As a result of an informal meeting held at Mainz in November 1958, about forty interested persons from several European countries held an ALGOL implementation conference in Copenhagen in February 1959. A "hardware group" was formed for working cooperatively right down to the level of the paper tape code. This conference also led to the publication by Regnecentralen, Copenhagen, of an ALGOL *Bulletin*, edited by Peter Naur, which served as a forum for further discussion. During the June 1959 ICIP Conference in Paris several meetings, both formal and informal ones, were held. These meetings revealed some misunderstandings as to the intent of the group which was primarily responsible for the formulation of the language, but at the same time made it clear that there exists a wide appreciation of the effort involved. As a result of the discussions it was decided to hold an international meeting in January 1960 for improving the ALGOL language and preparing a final report. At a European ALGOL Conference in Paris in November 1959 which was attended by about fifty people, seven European representatives were selected to attend the January 1960 Conference, and they represent the following organizations: Association Francaise de Calcul, British Computer Society, Gesellschaft fur Angewandte Mathematik und Mechanik, and Nederlands Rekenmachine Genootschap. The seven representatives held a final preparatory meeting at Mainz in December 1959.

Meanwhile, in the United States, anyone who wished to suggest changes or corrections to ALGOL was requested to send his comments to the ACM *Communications* where they were published. These comments then became the basis of consideration for changes in the ALGOL language. Both the SHARE and USE organizations established ALGOL working groups, and both organizations were represented on the ACM Committee on Programming Languages. The ACM Committee met in Washington in November 1959 and considered all comments on ALGOL that had been sent to the ACM *Communications*. Also, seven representatives were selected to attend the January 1960 international conference. These seven representatives held a final preparatory meeting in Boston in December 1959.

January 1960 Conference

The thirteen representatives,[3] from Denmark, England, France, Germany, Holland, Switzerland, and the United States, conferred in Paris from January 11 to 16, 1960.

Prior to this meeting a completely new draft report was worked out from the preliminary report and the recommendations of the preparatory meetings by Peter Naur and the conference adopted this new form as the basis for its report. The Conference then proceeded to work for agreement on each item of the report. The present report represents the union of the Committee's concepts and the intersection of its agreements.

As with the preliminary ALGOL report, three different levels of language are recognized, namely a Reference Language, a Publication Language and several Hardware Representations.

REFERENCE LANGUAGE

1. It is the working language of the committee.
2. It is the defining language.

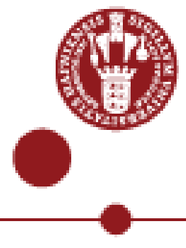
[1] Preliminary report—International Algebraic Language, *Comm. Assoc. Comp. Mach. 1*, No. 12 (1958), 8.

[2] Report on the Algorithmic Language ALGOL by the ACM Committee on Programming Languages and the GAMM Committee on Programming, edited by A. J. Perlis and K. Samelson, *Numerische Mathematik* Bd. *1*, S. 41-60 (1959).

[3] William Turanski of the American group was killed by an automobile just prior to the January 1960 Conference.

What Donald Knuth saw

In 1967, Donald E. Knuth (who was also originally a mathematician) published a paper that pointed out a number of problems that still existed in ALGOL 60.



What does this procedure return?

```
integer procedure awkward
begin comment x is a global variable
x := x+1
awkward := 3 end awkward
```

What does this procedure return?

```
integer procedure awkward
begin comment x is a global variable
x := x+1
awkward := 3 end awkward
```

The ALGOL 60 report does not tell us if side effects are allowed in procedures, nor does it tell us if expressions are to be evaluated from left to right or from right to left! In particular, what is the value of `awkward+x`?

Morale

It is important to have an approach to descring program behaviour that is

- independent of implementations
- can be used as a specification of possible future implementations
- is easy to check for completeness (did we define everything?)
- is precise and unambiguous
- allows us to reason mathematically about program behaviour

Approaches to formal semantics

In the late 1960s, Christopher Strachey and Dana Scott came up with the first approach to giving a precise mathematical description of programming languages. This was the approach known as denotational semantics.

In the early 1980s, Gordon Plotkin and Robin Milner invented the approach known as structural operational semantics. This is the approach that I will consider here!

There are also other approaches to semantics such as axiomatic semantics (due to Hoare) and algebraic semantics (due to Goguen and others).

All these approaches are not rivals, simply different approaches to describing different aspects of program behaviour.

Static and dynamic semantics

We often make the distinction that

Dynamic semantics describes program behaviour – what happens at run-time

Static semantics describes types and internal dependencies beyond simply syntax – what is checked at compile-time

If we see things in this light, type systems are also part of the study of program semantics. Personally, I think of the study of type systems rather as a closely related topic.

Learning goals

- To understand and be able to explain the justification for the study of formal semantics and type systems
- To be able to write syntax-directed inference rules in the setting of structural operational semantics and type systems
- To be able to read and apply inference rules in the setting of structural operational semantics and type systems
- To be able to use the fundamental terminology of structural operational semantics and type systems

Part II

Abstract syntax

Semantics needs syntax!

If we want to specify how programs behave, we need a precise understanding of what programs look like.

Abstract syntax is our friend.

Syntactic categories and formation rules

The structure of syntactic terms
(≠ syntactic sugar)

- Syntactic categories
  - Expressions, statements, declarations ...
- Formation rules
  (structure of syntactic terms)

A tiny functional language

Syntactic categories

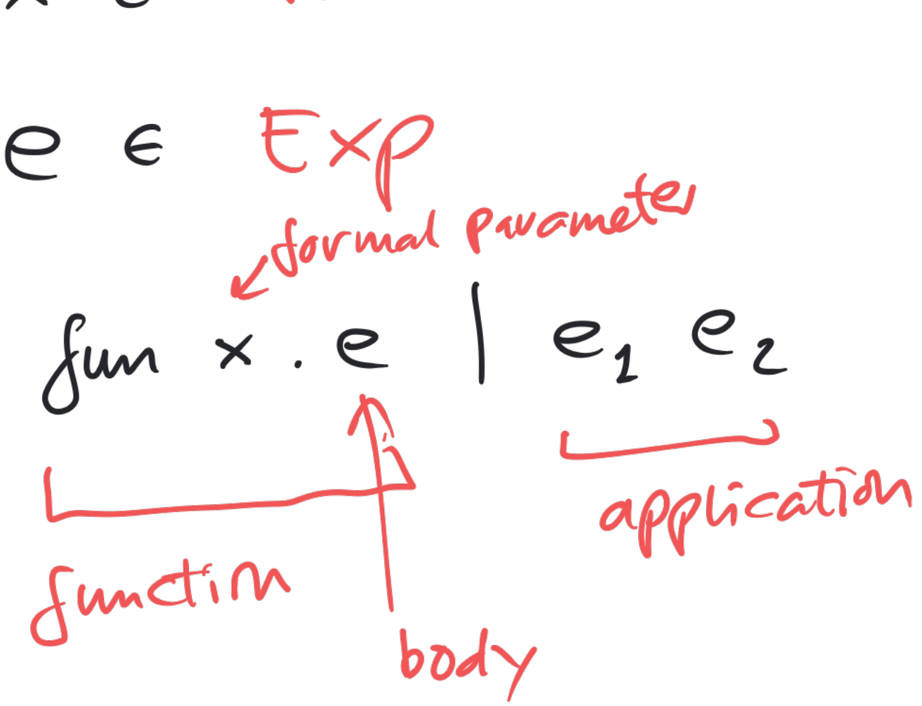
$n \in$ Num

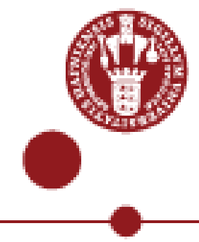
$x \in$ Var

$e \in$ Exp

Formation rules:

$$e ::= n \mid x \mid e_1 + e_2 \mid \mathit{fun}\ x\ .\ e \mid e_1\ e_2$$





Part III

Type systems

A type system for our functional language

type environment

$\tau \vdash e : t$

type

type judgement

τ : Var $\longrightarrow$ Types

$\tau[x \mapsto t]$

τ with the binding $x \mapsto t$ added

Formation rules for types:

$t ::= \text{int} \mid t_1 \longrightarrow t_2$

function type

Valid type judgements

- defined by a collection of inference rules

The type rules (syntax-directed)

[1 rule per syntactic construct]

$$e ::= n \mid x \mid e_1 + e_2 \mid e_1\, e_2 \mid \text{fun } x : t_1 . e$$

[Num] $\quad \tau \vdash n : \text{int}$

[Var] $\quad \tau \vdash x : t \quad \text{if } \tau(x) = t$

$$\text{[Plus]}\quad \frac{\tau \vdash e_1 : \text{int} \qquad \tau \vdash e_2 : \text{int}}{\tau \vdash e_1 + e_2 : \text{int}}$$

$$\text{[App]}\quad \frac{\tau \vdash e_1 : t_1 \to t_2 \qquad \tau \vdash e_2 : t_1}{\tau \vdash e_1\ e_2 : t_2}$$

we can mention x in here!

$$\text{[Fun]}\quad \frac{\tau[x \mapsto t_1] \vdash e : t_2}{\tau \vdash \text{fun } x : t_1 \to e : t_1 \to t_2}$$

$$\tau = [x \mapsto \mathrm{nt}]$$

$$\dfrac{\dfrac{\tau(x) = \mathrm{nt}}{\tau \vdash x : \mathrm{nt}}\,[\mathrm{Var}] \qquad \tau \vdash 7 : \mathrm{nt}\,[\mathrm{Num}]}{\tau \vdash x + 7 : \mathrm{nt}}\,[\mathrm{Plus}]$$

type derivation

Part IV

Structural operational semantics

Structural operational semantics

- Program behaviour as transitions
- Define the set of valid transitions

Two approaches

- Big-step semantics

A transition models an entire computation

- Small-step semantics

Often a good choice

A transition models a single step of a large computation

Big-step semantics of our functional language

semantic environment

value

$$\rho \vdash e \rightsquigarrow v$$

$$\rho : Var \longrightarrow Values$$

$$Values = \mathbb{Z} \cup Closures$$

(represent function values)

formal parameter

$fun\ x.e$	value	(x, e, ρ)	closure

body

bindings known

Transition rules

[Num] $$\rho \vdash n \rightsquigarrow n$$

[Var] $$\rho \vdash x \rightsquigarrow v \quad \text{if} \quad \rho(x) = v$$

[Plus] $$\frac{\rho \vdash e_1 \rightsquigarrow v_1 \quad \rho \vdash e_2 \rightsquigarrow v_2}{\rho \vdash e_1 + e_2 \rightsquigarrow v} \quad \text{where } v = v_1 + v_2$$

[Fun]

$$\rho \vdash \mathit{fun}\ x.e \rightsquigarrow (x, e, \rho)$$

[App]

$$\frac{\rho \vdash e_1 \rightsquigarrow (x, e_3, \rho_2) \qquad \rho \vdash e_2 \rightsquigarrow v \qquad \rho[x \mapsto v] \vdash e_3 \rightsquigarrow w}{\rho \vdash e_1\ e_2 \rightsquigarrow w}$$

CALL-BY-VALUE

Part V

Structural operational semantics (again)

An imperative language

Syntactic categories

$n \in \mathbb{N}um$

$s \in Stm$

$s ::= skip \mid s_1 ; s_2 \mid x := n \mid x{+}{+} \mid x{-}{-}$

$\mid$ if x then s_1 else s_2 $\mid$ while x do s

Programs

$P ::= S$

Input : in variable X

Output : in variable Y

Binding model

Transitions are of the form

$$\sigma \vdash s \rightsquigarrow \sigma'$$

$\sigma : \text{Var} \longrightarrow \text{Values}$

stores

(store the values of variables)

Transition rules for statements

[skip] $\sigma \vdash \text{skip} \rightsquigarrow \sigma$

[Ass] $\sigma \vdash x := n \rightsquigarrow \sigma[x \mapsto n]$

[Inc] $\sigma \vdash x{+}{+} \rightsquigarrow \sigma[x \mapsto v_x + 1]$

where $\sigma(x) = v_x$

[Dec] $\sigma \vdash x{-}{-} \rightsquigarrow \sigma[x \mapsto v_x - 1]$

where $\sigma(x) = v_x$

[comp]

$$\frac{\sigma \vdash s_1 \rightsquigarrow \sigma'' \qquad \sigma'' \vdash s_2 \rightsquigarrow \sigma'}{\sigma \vdash s_1 ; s_2 \rightsquigarrow \sigma'}$$

[if-false]

$$\frac{\sigma \vdash s_2 \rightsquigarrow \sigma'}{\sigma \vdash \text{if } x \text{ then } s_2 \text{ else } s_2 \rightsquigarrow \sigma'}$$

$\sigma(x) = v$ where $v = 0$

[if-true]

$$\frac{\sigma \vdash s_2 \rightsquigarrow \sigma'}{\sigma \vdash \text{if } x \text{ then } s_1 \text{ else } s_2 \rightsquigarrow \sigma'}$$

where $\sigma(x) = v \;,\; v > 0$

[while-false]

$$\sigma \vdash \text{while } x \text{ do } s \rightsquigarrow \sigma \qquad \text{where } \sigma(x) = 0$$

[while-true]

$$\frac{\sigma \vdash s \rightsquigarrow \sigma'' \qquad \sigma'' \vdash \underline{\text{while } x \text{ do } S} \rightsquigarrow \sigma'}{\sigma \vdash \underline{\text{while } x \text{ do } S} \rightsquigarrow \sigma'} \qquad \text{where } \sigma(x) > 0$$

$$\frac{\sigma \vdash \text{while } b \text{ do } S \rightsquigarrow \sigma' \qquad \sigma' \vdash \text{while } b \text{ do } S \rightsquigarrow \sigma''}{\sigma \vdash \text{while } b \text{ do } S \rightsquigarrow \sigma''}$$

$$\frac{}{\sigma \vdash \text{while } b \text{ do } S \rightsquigarrow \sigma}$$

Programs ?

$$\frac{\sigma[x \mapsto v] \vdash s \longrightarrow \sigma'}{\sigma[x \mapsto v] \vdash s \rightsquigarrow \sigma'} \quad \text{where } \sigma'(y) = v'$$

Part VI

More about static semantics

Extended imperative language with goto !

$$S ::= \text{skip} \mid x := n \mid x{+}{+} \mid x{-}{-}$$
$$\mid \text{if } x \text{ then } s_2 \text{ else } s_2 \mid \ell : \mid \text{goto } \ell$$

- No ℓ is defined more than once
- Every goto ℓ must have some ℓ given somewhere

New semantics

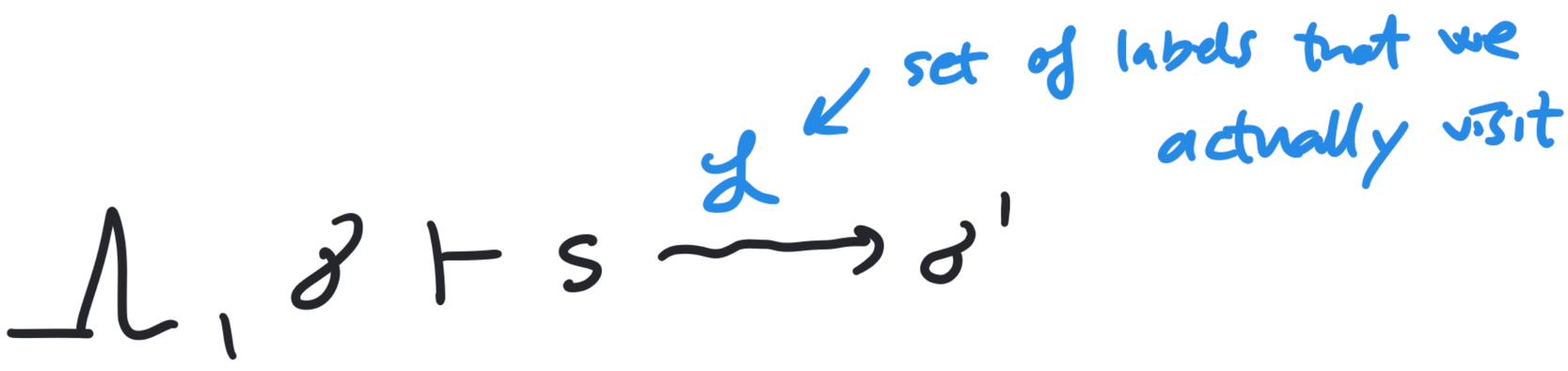
$$\Lambda, \sigma \vdash s \longrightarrow \sigma'$$

Λ: label environment

$$\frac{\Lambda, \sigma \vdash \Lambda\, \ell \longrightarrow \sigma'}{\Lambda, \sigma \vdash \text{goto } \ell \longrightarrow \sigma'}$$

↙ lookup the statement bound to ℓ

Record by labelling the semantics
what labels we actually visit



Approximating $\mathcal{L}$

$\vdash s \triangleleft \lambda$] defined by rules!

(If we execute the statement s, the labels that we goto along the way are at most the labels in the set λ)

Some rules (of a static semantics)

$$\vdash \texttt{skip} \triangleleft \emptyset$$

$$\vdash \ell : \triangleleft \{\ell\}$$

(no label can be defined more than once)

$$\frac{\vdash s_1 \triangleleft \lambda_1 \qquad \vdash s_2 \triangleleft \lambda_2 \qquad \lambda_1 \cap \lambda_2 = \emptyset}{\vdash \texttt{if } x \texttt{ then } s_1 \texttt{ else } s_2 \triangleleft (\lambda_1 \cup \lambda_2)}$$

Part VII

How are type systems and operational semantics related?

The type system for our functional language

meets

The big-step semantics of our functional language

Theorem (type safety)

If $\tau_\emptyset \vdash e : t$

then $\S_\emptyset \vdash e \rightsquigarrow v$

The big-step semantics of the goto language

meets

The static semantics of the goto language

(safety of our static semantics)

Theorem

If $\vdash s \triangleleft \lambda$ / λ is an overapproximation of $\mathcal{L}$

and $\sigma_{\emptyset}, \Lambda \vdash s \xrightarrow{\mathcal{L}} \sigma'$

then $\mathcal{L} \subseteq \lambda$

- We can formulate type safety precisely!
- We can formulate correctness of static analysis precisely

- "Running the type system backwards" (building a derivation tree)
  = type checking
- "Running the big-step semantics"
  = an interpreter!