Introduction

CS 152 (Spring 2024)

Harvard University

Tuesday, January 23, 2024

Programming Languages

- More than a catalog of languages and what they can be used for.
- In this class: foundations of programming languages, the underlying concepts and principles that go into designing and implementing programming languages.
- How can you learn new languages? How can you design effective languages?

Why?

- give you the concepts to more easily learn new languages
- ... and to design and implement new languages
- golden age of PL
- elegant math

Aspects of a Language

syntax	the structure of its programs
semantics	the meaning of its programs

A formal semantics...

- can be simpler than an implementation, more precise than intuition
- can answer: is this implementation correct
- supports the definition of analyses and transformations
  - prove properties about the language
  - prove properties about programs written in the language
- promotes better language design
  - better understand impact on design decisions
  - apply semantic insights to improve language

Cool: Type safety

Example: Rust

Rust is memory safe (no deferencing of null pointers, no dangling pointers), but performance is comparable to C and C++. Lots of memory checking is done statically. Achieves this using a sophisticated type system, with parametric polymorphism and linear types. All at compile time, with no run-time overhead.

Cool: Certified compilers

- Formal proof that the native code output by CompCert has the same semantics as the original C program.
- Researchers found zero bugs in the verified part of CompCert vs hundreds of bugs in LLVM and GCC.

Cool: Program Synthesis

Cool: Program Verification

Cool: Differentiable Programming

Cool: Probabilistic Programming

ToC

- semantics
- lambda calculus
- types
- reasoning about programs
- misc. topics

Semantics of Programming Languages

Give *mathematical meaning* to programs.

Why *mathematical*?

- Less ambiguous.
- More concise.
- Formal arguments.

Semantics

Styles of Semantics

Operational Semantics
Denotational Semantics
Axiomatic Semantics
Algebraic Semantics

Operational Semantics

- Small-Step
- Large-Step

Small-Step Operational Semantics

step from configuration to configuration:

$$c_0 \longrightarrow c_1 \longrightarrow \ldots \longrightarrow c_n$$

Large-Step Operational Semantics

one step from initial configuration to final answer:

$$c \Downarrow a$$

Denotational Semantics

interpret in mathematical domain

$$[\![\text{term}]\!] = \text{number}$$

$$[\![e_1 + e_2]\!] = [\![e_1]\!] + [\![e_2]\!]$$

$$\ldots$$

Axiomatic Semantics

$$\{Pre\}\ c\ \{Post\}$$

Algebraic Semantics

Abstract Syntax

Abstract Syntax

$$x, y, z \in \textbf{Var}$$

$$n, m \in \textbf{Int}$$

$$e \in \textbf{Exp}$$

Abstract Syntax

$$x, y, z \in \textbf{Var}$$

Var is the set of program variables (e.g., foo, bar, baz, i, etc.).

Abstract Syntax

$$n, m \in \textbf{Int}$$

Int is the set of constant integers (e.g., 42, -40, 7).

Abstract Syntax

$$e \in \mathbf{Exp}$$

Exp is the domain of expressions, which we specify using a BNF (Backus-Naur Form) grammar.

Simple Expressions

$$
\begin{aligned}
e ::= \; & x \\
\mid \; & n \\
\mid \; & e_1 + e_2 \\
\mid \; & e_1 \times e_2
\end{aligned}
$$

Abstract Syntax Tree

$$1 + 2 \times 3$$

Abstract Syntax Tree

$$1 + 2 \times 3$$

$$1 + (2 \times 3) \qquad (1 + 2) \times 3$$

Abstract Syntax Tree

$$1 + 2 \times 3$$

```
    +
   / \
  1   *
     / \
    2   3
```

$1 + (2 \times 3)$

$(1 + 2) \times 3$

Abstract Syntax Tree

$$1 + 2 \times 3$$

```
    +
   / \
  1   *
     / \
    2   3
```

$$1 + (2 \times 3)$$

```
      *
     / \
    +   3
   / \
  1   2
```

$$(1 + 2) \times 3$$

Def. and Use of Abstract Syntax

- in OCaml
- in Coq
- in Dafny