441: Programming Languages
MinML: A MINiMaL Functional Language
Dynamic Semantics

David Walker

Dynamic Semantics

The **dynamic semantics** of a language specifies how to execute programs written in that language.

Two general approaches:

1. **Machine-based**: describe execution in terms of a mapping of the language onto an (abstract or concrete) machine.

2. **Language-based**: describe execution entirely in terms of the language itself.

Machine-Based Models

Historically, machine-based approaches have dominated.

- Assembly languages.
- Systems languages such as C and its derivatives.

Such languages are sometimes called **concrete** languages because of their close association with the machine.

Machine-Based Models

Advantages:

- Specifies meanings of data types in terms of machine-level concepts.

- Facilitates low-level programming, *e.g.*writing device drivers.

- Supports low-level "hacks" based on the quirks of the target machine.

Machine-Based Models

Disadvantages:

- Requires you to understand how a language is compiled.
- Inhibits portability.
- Run-time errors (such as “bus error”) cannot be understood in terms of the program, only in terms of how it is compiled an executed.

Language-Based Models

Define execution behavior entirely at the level of the language itself.

"Computation by calculation."

No need to specify implementation details.

Such languages are sometimes called **abstract** languages because they abstract from machine-specific details.

Language-Based Models

Advantages:

- Inherently portable across platforms.
- Semantics is defined entirely in terms of concepts within the language.
- No mysterious (implementation-specific) errors to track down.

Language-Based Models

Disadvantages:

- Cannot take advantage of machine-specific details.

- Can be more difficult to understand complexity (time and space usage).

Machine- vs.Language-Based Models

Language-based models will dominate in the future.

- Low-level programming is a vanishingly small percentage of the mix.
- Emphasis on bit-level efficiency is almost always misplaced.
- Portability matters much more than efficiency.

Dynamic Semantics of MinML

We'll define the dynamic semantics of MinML using a technique called **structured operational semantics (SOS)**.

- Define a **transition relation** $p \mapsto p'$ between programs.
- A transition consists of execution of a single **instruction**.
- Rules determine which instruction to execute next.
- There are no transitions from **values**.

Values

The set of **values** is inductively defined by the following rules:

$$\frac{x \text{ var}}{x \text{ value}} \qquad \frac{n \text{ number}}{n \text{ value}}$$

$$\frac{}{\texttt{true value}} \qquad \frac{}{\texttt{false value}}$$

$$\frac{\tau_1 \text{ type} \quad \tau_2 \text{ type} \quad f \text{ var} \quad x \text{ var} \quad e \text{ expr}}{\texttt{fun}\, f\ \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2 = e \text{ value}}$$

Primitive Instructions

First, we define the **primitive instructions** of MinML. These are the atomic transition steps.

- Primitive operations on numbers.
- Conditional branch when the test is either `true` or `false`.
- Application of a recursive function to an argument value.

Primitive Instructions

Addition of two numbers:

$$\frac{(n = n_1 + n_2)}{\texttt{+}(n_1, n_2) \mapsto n}$$

Equality test:

$$\frac{}{\texttt{=}(n_1, n_2) \mapsto \begin{cases} \texttt{true} & n_1 = n_2 \\ \texttt{false} & n_1 \neq n_2 \end{cases}}$$

Primitive Instructions

Conditional branch:

$$\frac{}{\texttt{if}_\tau\,\texttt{true}\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,\texttt{fi} \mapsto e_1}$$

$$\frac{}{\texttt{if}_\tau\,\texttt{false}\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,\texttt{fi} \mapsto e_2}$$

Primitive Instructions

Application of a recursive function:

$$\frac{v \ \textsf{value} \quad v_1 \ \textsf{value} \quad (v = \texttt{fun}\, f\, (x : \tau_1) : \tau_2 \ = \ e)}{\texttt{apply}(v, v_1) \mapsto \{v, v_1/f, x\}e}$$

NB: we substitute the **entire** function expression for f in e!

This "unrolls" the recursion by ensuring that f refers to the function itself.

Primitive Instructions

The rule for function application "unrolls the recursion" during application.

- Substitute the argument value for the function's parameter in its body.

- Substitute the **function itself** for the "self" parameter of the function in its body.

This ensures that calls to `f` (it's "local name") in the body are applications of f (the function itself).

Search Rules

Second, we specify the next instruction to execute by a set of **search rules**.

These rules specify the **order of evaluation** of MinML expressions: which instruction is to be executed next?

Assembly language programs are linear sequences of instructions; for these languages a simple counter (the PC) determines the next instruction.

For more structured languages such as MinML more complex rules are required.

Search Rules

The arguments of the primitive operations are evaluated left-to-right:

$$\frac{e_1 \mapsto e_1'}{\mathsf{+}(e_1, e_2) \mapsto \mathsf{+}(e_1', e_2)}$$

$$\frac{v_1 \ \mathsf{value} \quad e_2 \mapsto e_2'}{\mathsf{+}(v_1, e_2) \mapsto \mathsf{+}(v_1, e_2')}$$

Search Rules

For conditionals we evaluate the test expression:

$$\frac{e \mapsto e'}{\begin{array}{c}\texttt{if}_\tau\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \\ \mapsto \\ \texttt{if}_\tau\, e'\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi}\end{array}}$$

Search Rules

Applications are evaluated left-to-right: first the function, then the argument.

$$\frac{e_1 \mapsto e_1'}{\texttt{apply}(e_1, e_2) \mapsto \texttt{apply}(e_1', e_2)}$$

$$\frac{v_1 \ \textsf{value} \quad e_2 \mapsto e_2'}{\texttt{apply}(v_1, e_2) \mapsto \texttt{apply}(v_1, e_2')}$$

Multi-step Evaluation

The relation $e \mapsto^* e'$ is inductively defined by the following rules:

$$\frac{}{e \mapsto^* e} \qquad \frac{e \mapsto e' \quad e' \mapsto^* e''}{e \mapsto^* e''}$$

That is, $e \mapsto^* e'$ iff $e = e_0 \mapsto e_1 \mapsto \cdots \mapsto e_n = e'$ for some $n \geq 0$.

Example Execution

Suppose that f is the expression

```
fun f(n:int):int is if n=0 then 1 else n*f(n-1)
```

Consider the evaluation of `apply(`f`,3)`.

This a primitive instruction, which we execute:

$$\texttt{apply(}f\texttt{,3)} \;\mapsto\; \texttt{if 3=0 then 1 else 3*}f\texttt{(3-1)}$$

We have substituted `3` for `n` and f for `f` in the body of the function.

Example Execution

We now evaluate the test and branch:

$$
\begin{array}{lll}
\texttt{if 3=0 then 1 else 3*}f\texttt{(3-1)} & \mapsto & \texttt{if false then 1 el} \\
 & \mapsto & \texttt{3*}f\texttt{(3-1)} \\
 & \mapsto & \texttt{3*}f\texttt{(2)} \\
 & \mapsto & \texttt{3*(if 2=0 then 1 e} \\
 & \mapsto^{*} & \texttt{3*2*}f\texttt{(1)} \\
 & \mapsto^{*} & \texttt{3*2*1*}f\texttt{(0)} \\
 & \mapsto^{*} & \texttt{3*2*1*1} \\
 & \mapsto^{*} & \texttt{6}
\end{array}
$$

Induction on Evaluation

Since one-step evaluation is inductively defined, there is an associated principle of induction, called **induction on evaluation**.

To prove that $e \mapsto e'$ implies $P(e, e')$ for some property P, it suffices to prove that P is closed under the rules of evaluation.

1. $P(e, e')$ holds for each of the instruction axioms.

2. Assuming P holds for each of the premises of a search rule, show that it holds for the conclusion as well.

Induction on Evaluation

Similarly, multi-step evaluation is inductively defined, and hence there is an associated principle of induction, called **induction on the steps of evaluation**.

To show that $e \mapsto^* e'$ implies $P(e, e')$, it suffices to show

1. If $P(e, e)$, *i.e.*that P is **reflexive**.

2. If $e \mapsto e' \mapsto^* e''$ and $P(e', e'')$, then $P(e, e'')$. This is called **closure under reverse evaluation**.

Elementary Properties of Evaluation

Proposition 1 (Values Irreducible)
If v value *then there is no* e *such that* $v \mapsto e$ *(i.e.,* $v \not\mapsto$*).*

Proof: By inspection of the rules.

1. No instruction is a value.

2. No search rule applies to a value.

■

Elementary Properties of Evaluation

Proposition 2 (Determinacy)
For every e there exists at most one e' such that $e \mapsto e'$.

Proof: By induction on the structure of e, making use of the irreducibility of values to handle apparent overlapping cases. For example, the first application rule can apply only if the first argument is **not** a value, by the previous proposition. ■

Elementary Properties of Evaluation

Every expression has at most one value.

Corollary 3 (Determinacy of Values)
For any e there exists at most one v such that $e \mapsto^* v$.

In other words, the relation $\mapsto^*$ is a **partial function**.

Stuck States

Not every irreducible expression is a value!

$$\texttt{if 7 then 1 else 2} \not\mapsto$$

$$\texttt{true+false} \not\mapsto$$

$$\texttt{0(1)} \not\mapsto$$

Observe that all are ill-typed.

An expression e that is not a value, but for which there exists no e' such that $e \mapsto e'$ is said to be **stuck**.

Safety: **all** stuck expressions are ill-typed. Equivalently, well-typed expressions do not get stuck.

Summary

MinML is a **language-based** model of computation.

- Evaluation is defined on the expressions themselves.

- No mention of a mapping onto a machine.

Summary

The dynamic semantics of MinML is given using **structured operational semantics**.

- Rules for primitive instructions.

- Rules for determining the next instruction.