1 Introduction

In order to talk rigoursously about how programs behave and and reason about programs we need a concise and formal specification of a programming language. Such a specification consists of three steps: 1) the grammar of the language 2) operational dynamic semantics 3) static semantics (= type system).

We will consider a small functional language called Mini-ML, which forms the basis of programming languages such as SML, OCaml, or Haskell. It is worth stressing that any techniques we describe here are general enough that they apply to other languages such as Java as well.

2 Inductive definitions of expressions

We will start with an inductive defintion of well-formed expressions. Our language will have numbers and some basic arithmetic operations, booleans and if-statements, functions (recursive with names, and nameless), applications and variables. In addition, we will consider a let-construct.

$$\begin{array}{lll} \text{expressions } e & ::= & n \mid e_1 + e_2 \mid e_1 = e_2 \mid e_1 * e_2 \mid e_1 < e_2 \\ & & \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \\ & & x \mid \mathsf{fun}\ f(x) = e \mid e_1\ e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \ \mathsf{end} \end{array}$$

This grammar inductivly specifies well-formed expressions. To illustrate, we consider some well-formed and some ill-formed expressions.

Examples of well-formed expressions

- $3 + (2 + 4)$
- $\mathsf{true} + (2 + 4)$
- $\mathsf{fun}\ sum(x) = \mathsf{if}\ (x = 10)\ \mathsf{then}\ x\ \mathsf{else}\ x + sum\ (x + 1)$
- $\mathsf{let}\ z = \mathsf{if}\ true\ \mathsf{then}\ 2\ \mathsf{else}\ 43\ \mathsf{in}\ z + 123\ \ \mathsf{end}$
- $\mathsf{let}\ z = \mathsf{if}\ 7\ \mathsf{then}\ 2\ \mathsf{else}\ 43\ \mathsf{in}\ z + \mathsf{false}\ \ \mathsf{end}$

Examples of ill-formed expressions

- fun $sum(x) =$ if $(x = 10)$ then x else
- let $z =$ if $true$ then 2 else 43 in $+ 123$ end

Note that the grammar only tells us when expressions are well-formed. It does not tell us whether an expression is in fact meaningfull.

3 Big-step semantics

Next, we define the operational semantics for Mini-ML. We will start with a big-step semantics.

Evaluation judgement:

$$e \Downarrow v \quad \text{Expression } e \text{ evaluates to a final value } v$$

We will define evaluation inductively on the structure of expressions e. If we have already reached a value, then we will return this value. If we have an expresssion e which is not yet a value, we will proceed by evaluating the sub-expression of e. We will use a substitution model to describe evaluation. This has several advantages. It will give a high-level, simple description of evaluation, and will make it easier to reason about the high-level aspects of programs. It is not purely interesting from a theoretical point of view, but also has some practical applications, since it forms the basis of compiler optimizations called inlining.

Numbers and arithmetic expressions We start by defining the evaluation rules for numbers and arithmetic operations. They are straightforward.

$$\frac{}{n \Downarrow n}\ \text{B-NUM} \qquad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad \text{primop}(v_1, v_2) = v}{e_1 \ \mathsf{op}\ e_2 \Downarrow v}\ \text{B-OP}$$

Note that instead of defining an inference rule for each arithmetic operation $+, *, -, =$ we define the rule for evaluating arithmetic expressions generically by $e_1 \mathsf{op} e_2$ where $\mathsf{op} \in \{+, *, =, -\}$.

Booleans and if-expression The evaluation rules for booleans and if-statement are straightforward.

$$\frac{}{\mathsf{true} \Downarrow \mathsf{true}}\ \text{B-TRUE} \qquad \frac{}{\mathsf{false} \Downarrow \mathsf{false}}\ \text{B-FALSE} \Downarrow$$

$$\frac{e \Downarrow \mathsf{true} \quad e_1 \Downarrow v}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow v}\ \text{B-IFT} \qquad \frac{e \Downarrow \mathsf{false} \quad e_2 \Downarrow v}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow v}\ \text{B-IFF}$$

There are two rules for evaluating if-expressions. If the guard e evaluates to true, we will evaluate the first branch (rule B-IFT), and if the guard e evaluates to false, we will evaluate the second one (rule B-IFF).

Functions and function application Functions are considered first-class values, and hence will evaluate to themselves (see rule B-FUN). Evaluation of function application ($e_1\ e_2$) is done in three steps. First, we evaluate e_1 which will (hopefully!) yield a function fun $f(x) = e$. In addition, we evaluate e_2 to obtain some value v_2. Finally, we need to evaluate the function body e where we have replaced any occurrence of x with v_2. However just evaluating $[v_2/x]e$ is not enough, since our function fun $f(x) = e$ may be recursive. To simply model recursive evaluation, we will also replace any occurrence of f in e with the actual function definition fun $f(x) = e$.

$$\frac{e_1 \Downarrow \mathsf{fun}\ f(x) = e \quad e_2 \Downarrow v_2 \quad [v_2/x][\mathsf{fun}\ f(x) = e\ /f]e \Downarrow v}{\mathsf{apply}\ (e_1,\ e_2) \Downarrow v}\ \text{B-APP}$$

$$\frac{}{\mathsf{fun}\ f(x) = e\ \Downarrow \mathsf{fun}\ f(x) = e}\ \text{B-FUN}$$

Let-expression Let-expressions are similar to functions and function application. In fact let $x = e_1$ in e_2 end can be defined as (fun $f(x) = e_2$) e_1. As a consequece, the evaluation rule for let-expressions is derived from the evaluation rule for function application. After evaluating e_1 to some value v_1, we will continue evaluating $[v_1/x]e_2$.

Let-statement

$$\frac{e_1 \Downarrow v_1 \quad [v_1/x]e_2 \Downarrow v}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \ \mathsf{end} \Downarrow v}\ \text{B-LET}$$

Remark Note that the evaluation rules do not impose an order in which premises need to be evaluated. For example, when evaluating $e_1 \mathsf{op} e_2$, we can first evaluate e_1 and then e_2 or the other way round. The rule just specifies that both subexpressions need to be evaluated. Similarly, in the rule for function application, we could first evaluate e_2 and then evaluate e_1. Big-step evaluation provides a high-level description of the operational semantics. Although it abstract over the order in which we evaluate sub-expressions, the evaluation rules enforce a call-by-value strategy in the rules B-LET and B-APP. For example in the B-LET-rule, we evaluate e_1 and then substitute the value of e_1 into expression e_2 which is then evaluated. (Question: How would you change these rules to allow call-by-name?)

Example Next, we give an example of an evaluation derivation to illustrate the use of the evaluation rules.

4 Properties

Our operational semantics has various interesting properties.

Value soundness If $e \Downarrow v_1$ then v_1 is a value.

Determinacy If $e \Downarrow v_1$ and $e \Downarrow v_2$ then $v_1 = v_2$.

We can prove these properties by structureal induction on the inference rules for evaluation.

Theorem 4.1 (Determinacy) *If $e \Downarrow v_1$ and $e \Downarrow v_2$ then $v_1 = v_2$.*

Proof: Proof by structural induction on the first derivation $e \Downarrow v_1$. We will refer to the first derivation as $\mathcal{D} = e \Downarrow v_1$ and consider each possible derivation for $\mathcal{D}$ in turn.

Case: $\mathcal{D} = \dfrac{}{n \Downarrow n}\ \textsc{b-num}$

By assumption $n \Downarrow v_2$. Since there is only the evaluation rule B-NUM can possibly be applied, we know that $v_2 = n$. Therefore, $n = n$.

All the cases for B-TRUE, B-FALSE and B-FUN will follow the same pattern. $\Box$

Arithmetic operations

$$\frac{t_i \longrightarrow t_i'}{op(v_1, \ldots, t_i, \ldots, t_n) \longrightarrow op(v_1, \ldots, t_i', \ldots, t_n)} \text{ E-OPARG}$$

$$\frac{\text{by primop } o}{op(v_1, \ldots, v_n) \longrightarrow v} \text{ E-OPVAL}$$

If-statement

$$\frac{}{\mathsf{if\ true\ then}\ t_1\ \mathsf{else}\ t_2 \longrightarrow t_1} \text{ E-IF-TRUE}$$

$$\frac{}{\mathsf{if\ false\ then}\ t_1\ \mathsf{else}\ t_2 \longrightarrow t_2} \text{ E-IF-FALSE}$$

$$\frac{t \longrightarrow t'}{\mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \longrightarrow \mathsf{if}\ t'\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2} \text{ E-IF}$$

Function application

$$\frac{t_1 \longrightarrow t_1'}{\mathsf{apply}\ (t_1,\ t_2) \longrightarrow \mathsf{apply}\ (t_1',\ t_2)} \text{ E-APP}$$

$$\frac{t_2 \longrightarrow t_2'}{\mathsf{apply}\ (v_1,\ t_2) \longrightarrow \mathsf{apply}\ (v_1,\ t_2')} \text{ E-APPVT}$$

$$\frac{v_1 = \mathsf{fun}\ f(x) = \tau_1\ \tau_2 t}{\mathsf{apply}\ (v_1,\ v_2) \longrightarrow [v1/f, v_2/x]t} \text{ E-APPVV}$$

Let-statement

$$\frac{t_1 \longrightarrow t_1'}{\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2\ \ \mathsf{end} \longrightarrow \mathsf{let}\ x = t_1'\ \mathsf{in}\ t_2\ \ \mathsf{end}} \text{ E-LET}$$

$$\frac{}{\mathsf{let}\ x = v_1\ \mathsf{in}\ t_2\ \ \mathsf{end} \longrightarrow [v_1/x]t_2} \text{ E-LETV}$$

Figure 1: One-step evaluation semantics fom MinML