15-312 Lecture on Evaluation and Functions

Diagrammatic View of the Evaluation Process

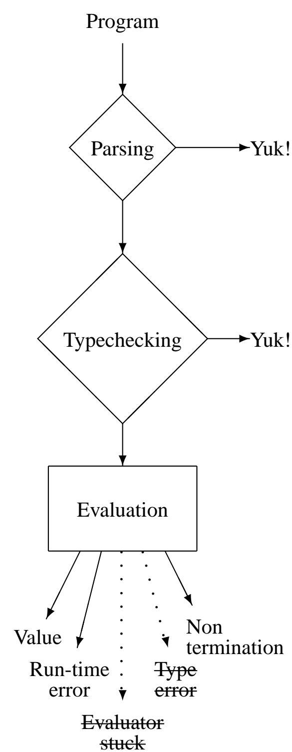

Figure 1: The Evaluation Process

Once a program or expression has successfully been parsed and typechecked, there can be 5 outcomes of evaluation (3 acceptable, 2 unacceptable):

1. A *value* is returned. This is typically the expected outcome of evaluation.

2. A *run-time error* is produced. By run-time error, we mean some dramatic event such as a division by zero or an integer overflow, which cannot generally be caught by the typechecker. We have seen that run-time errors can be treated as a very special form of returned values. We will see that we can also treat them using exceptions.

3. *Non-termination*: sometimes this results from a bug in the program, but it can also be a design decision for reactive systems such as a user interface (think Windows, or your cell phone).

~~4.~~ A *type error* is produced. This should never happen of a well-designed language. A correct proof of a type preservation theorem prevents this.

~~5.~~ The evaluator gets *stuck* and does not know what to do. This can happen if a transition was forgotten. This too should never happen of a well-designed language. A correct proof of a progress theorem prevents this.

Relating Transition and Evaluation Semantics

The theorem you want to prove is as follows:

Theorem 1 *Let e be an expression, then $\mathcal{E} :: e \Downarrow v$ iff $\mathcal{T} :: e \mapsto^* v$ and $\mathcal{V} :: v \ \mathsf{val}$.*

The two directions are proved separately. Going right-to-left is fairly easy:

Lemma 2 *Let e be an expression. If $\mathcal{E} :: e \Downarrow v$, then there exist derivations $\mathcal{T} :: e \mapsto^* v$ and $\mathcal{V} :: v \ \mathsf{val}$.*

Proof: By induction on $\mathcal{E}$. Applications of the induction hypothesis correspond to unfolding $\mathcal{E}$ so that the resulting transition sequence that is one step shorter than what we want, and then extending it as needed to obtain $\mathcal{T}$. □

The other direction requires care because the transition semantics is defined on the basis of the single step transition judgment, while the desired result refers to the entire transition sequence. We circumvent this fact by means of the following lemma, which concentrates on the step transition judgment.

Lemma 3 *Given expressions e and e' and a derivation $\mathcal{V} :: v \ \mathsf{val}$, if $\mathcal{T} :: e \mapsto^* e'$ and $\mathcal{D} :: e' \Downarrow v$, then there is a derivation $\mathcal{E} :: e \Downarrow v$.*

Proof: By induction on $\mathcal{D}$. □

Then, the desired left-to-right direction of the theorem is a simple corollary.

Corollary 4 *Given an expression e, if $\mathcal{T} :: e \mapsto^* v$ and $\mathcal{V} :: v \ \mathsf{val}$, then there is a derivation $\mathcal{E} :: e \Downarrow v$.*

Proof: By induction on $\mathcal{T}$. □

Deconstructing ML Functions

Consider the usual ML declaration for the factorial function:

```
fun fact (n: int): int =
    if (n = 1)
      then 1
      else n * fact (n-1)
```

We will now see that it relies on at least 3 language features that we will study separately in this course.

1. **Name Binding:** A first thing that happens in the above declaration is that the name "`fact`" is bound and available in the code that follows. This is more evident if we expand "`fun`" into the declaration it actually abbreviates:

   ```
   val rec fact = fn (n: int) =>
                    if (n = 1)
                      then 1
                      else n * fact (n-1)
   ```

   We have already encountered this mechanism, although with a different syntax, as the "`let`" construct.

2. **Recursion:** The keyword "`rec`" indicates that this is a recursive declaration. We will define the semantics of recursion in a few lectures.

3. **Functions:** The keyword "`fn`" introduces a function, which is the main concept of the current lecture. This mnemonic ML keyword is often written using the Greek letter λ (lambda) in the semi-abstract syntax, and "`lam`" as an abstract syntax constructor.

In truth, what is commonly viewed as an ML function hides many more constructs, which we will examine in later parts of this course. A function accepting multiple arguments formally relies on *tuples*. In the above example, calling "`fact`" with a negative argument (or an argument that is too large) will eventually trigger an *exception*. And of course, only the greenest of beginners would write "`fact`" as we did, as *patterns* as so much more convenient to use:

```
fun fact (0: int): int = 0
  | fact (n)           = n * fact (n-1)
```

Macho ML programmers and inexperienced students will rely on *type reconstruction* to quickly whack this function as:

```
fun fact 0 = 0
  | fact n = n * fact (n-1)
```