

DYNAMICS

Alex Kavvos

Reading: PFPL, §5.1, 5.2

We have studied the statics—i.e. the concrete syntax and type system—for a rudimentary programming language of numbers and strings. It is now time to look into the computational behaviour—or **dynamics**—of programs.

We will set up a **transition system** that specifies the states of evolution of a program, beginning from some initial term of interest, and ending with a final **value**.

1 Values

What is the aim of a program? For now, we will assume that it is to **compute a value**. This is a rather functional way of looking at programming. In contrast, imperative languages seek to effect some change on the world (write in memory, print a value, etc.). We will study such languages later on.

We define the judgement $e \text{ val}$ by the following rules.

$$\frac{\text{VAL-NUM} \quad n \in \mathbb{N}}{\text{num}[n] \text{ val}}$$

$$\frac{\text{VAL-STR} \quad s \in \Sigma^*}{\text{str}[s] \text{ val}}$$

In other words, we will only accept numbers and strings as values, i.e. results of a computation. It is evident that

Proposition 1. If $e \text{ val}$ then either $\vdash e : \text{Num}$ or $\vdash e : \text{Str}$.

Thus, every value is a **closed** term: it is typable in a context with no free variables.

2 Transitions

We will define a relation $e_1 \mapsto e_2$ between closed terms by the following rules.

$$\frac{\text{D-PLUS} \quad n_1 + n_2 = n}{\text{plus}(\text{num}[n_1]; \text{num}[n_2]) \mapsto \text{num}[n]}$$

$$\frac{\text{D-PLUS-1} \quad e_1 \mapsto e'_1}{\text{plus}(e_1; e_2) \mapsto \text{plus}(e'_1; e_2)}$$

$$\frac{\text{D-PLUS-2} \quad e_1 \text{ val} \quad e_2 \mapsto e'_2}{\text{plus}(e_1; e_2) \mapsto \text{plus}(e_1; e'_2)}$$

$$\frac{\text{D-CAT} \quad s_1 ++ s_2 = s}{\text{cat}(\text{str}[s_1]; \text{str}[s_2]) \mapsto \text{str}[s]}$$

$$\frac{\text{D-CAT-1} \quad e_1 \mapsto e'_1}{\text{cat}(e_1; e_2) \mapsto \text{cat}(e'_1; e_2)}$$

$$\frac{\text{D-CAT-2} \quad e_1 \text{ val} \quad e_2 \mapsto e'_2}{\text{cat}(e_1; e_2) \mapsto \text{cat}(e_1; e'_2)}$$

$$\frac{\text{D-LEN} \quad |s| = n}{\text{len}(\text{str}[s]) \mapsto \text{num}[n]}$$

$$\frac{\text{D-LEN-1} \quad e \mapsto e'}{\text{len}(e) \mapsto \text{len}(e')}$$

$$\frac{\text{D-LET}}{\text{let}(e_1; x. e_2) \mapsto e_2[e_1/x]}$$

Note: the rules for $\text{times}(e_1; e_2)$ are similar to those for $\text{plus}(e_1; e_2)$, and have been omitted.

Terms can be thought of as **states** of a transition system. The judgement $e_1 \mapsto e_2$ can be thought of as the relation that specifies the transitions between states. It is read as “ e_1 takes a step to e_2 .”

Some rules, like **D-PLUS**, perform computation; they are sometimes called **instruction transitions**.

Other rules, like **D-PLUS-1**, enable computation in a subterm; they are sometimes called **search transitions**. These determine the **order of evaluation**; e.g. here they force e_1 to be evaluated before e_2 in the term $\text{plus}(e_1; e_2)$.

Strictly speaking, transitions also require derivations like the one below.

$$\frac{\frac{\text{len}(\text{str}[\text{'asdf'}]) \mapsto \text{num}[4]}{\text{D-LEN}}}{\text{plus}(\text{len}(\text{str}[\text{'asdf'}]); \text{num}[1]) \mapsto \text{plus}(\text{num}[4]; \text{num}[1])} \text{D-PLUS-1}$$

In practice we write the transition, and underline the term to which an instruction transition is applied:

$$\text{plus}(\underline{\text{len}(\text{str}[\text{'asdf'}])}; \text{num}[1]) \mapsto \text{plus}(\text{num}[4]; \text{num}[1]) \quad (1)$$

3 Multi-step transitions

The transition (1) takes a step from a program to another program. It is clear that this second program is not yet a value: more transitions are needed to reach one.

$$\text{plus}(\underline{\text{len}(\text{str}[\text{'asdf'}])}; \text{num}[1]) \mapsto \underline{\text{plus}(\text{num}[4]; \text{num}[1])} \mapsto \text{num}[5] \quad (2)$$

A series of transitions is called a **transition sequence**.

We encapsulate transition sequences by defining the **reflexive transitive closure** of the relation \mapsto :

$$\frac{\text{D-MULTI-REFL}}{e \mapsto^* e} \qquad \frac{\text{D-MULTI-STEP} \quad \begin{array}{l} e \mapsto e' \quad e' \mapsto^* e'' \end{array}}{e \mapsto^* e''}$$

This relation is **reflexive**, as witnessed by the rule **D-MULTI-REFL** which postulates that $e \mapsto^* e$ for any e .

It is also **transitive**. However, this requires proof by induction:

Proposition 2. The rule $\frac{e_1 \mapsto^* e_2 \quad e_2 \mapsto^* e_3}{e_1 \mapsto^* e_3}$ is admissible.

It is also true that $e \mapsto^* e'$ if and only if there exists a transition sequence that proves this. In other words, there should exist pre-terms e_0, \dots, e_n (for $n \geq 0$) with

$$e = e_0 \mapsto \dots \mapsto e_n = e'$$

(This can be proven by induction, but is laborious and not very interesting.) For example, we have

$$\text{plus}(\text{len}(\text{str}[\text{'asdf'}]); \text{num}[1]) \mapsto^* \text{num}[5]$$

precisely because of the transition sequence (2). However, we **do not** have

$$\text{plus}(\text{len}(\text{str}[\text{'asdf'}]); \text{num}[1]) \mapsto \text{num}[5]$$

as this transition requires two steps of computation, not one.

4 Basic properties

If we are to think of values as final states of a computation, then there better be no transitions out of them.

Proposition 3 (Finality). If $e \text{ val}$ then there is no e' with $e \mapsto e'$.

The proof is by inspection. (Formally: by induction on $e \text{ val}$, and then inversion on $e \mapsto e'$.)

Every program computes a unique value. This is because the transition relation is **deterministic**.

Proposition 4 (Determinism). If $e \mapsto e_1$ and $e \mapsto e_2$ then $e_1 \equiv e_2$ (up to α -equivalence).

Hence, we are morally allowed to define $e \Downarrow v$ (“ e evaluates to value v ”) by

$$e \Downarrow v \stackrel{\text{def}}{=} e \mapsto^* v \wedge v \text{ val}$$

By **Proposition 4**, there is at most one v such that $e \Downarrow v$.