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 $e \ \mathsf{val}$ by the following rules.

$$\textsc{Val-Num}\ \frac{n \in \mathbb{N}}{\mathsf{num}[n]\ \mathsf{val}} \qquad \textsc{Val-Str}\ \frac{s \in \Sigma^*}{\mathsf{str}[s]\ \mathsf{val}}$$

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 val then either $\vdash e : \mathsf{Num}$ or $\vdash e : \mathsf{Str}$.

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

2 Transitions

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

$$\textsc{D-Plus}\ \frac{n_1 + n_2 = n}{\mathsf{plus}(\mathsf{num}[n_1]; \mathsf{num}[n_2]) \longmapsto \mathsf{num}[n]} \qquad \textsc{D-Plus-1}\ \frac{e_1 \longmapsto e_1'}{\mathsf{plus}(e_1; e_2) \longmapsto \mathsf{plus}(e_1'; e_2)} \qquad \textsc{D-Plus-2}\ \frac{e_1\ \mathsf{val} \quad e_2 \longmapsto e_2'}{\mathsf{plus}(e_1; e_2) \longmapsto \mathsf{plus}(e_1; e_2')}$$

$$\textsc{D-Cat}\ \frac{s_1 +\!\!+\, s_2 = s}{\mathsf{cat}(\mathsf{str}[s_1]; \mathsf{str}[s_2]) \longmapsto \mathsf{str}[s]} \qquad \textsc{D-Cat-1}\ \frac{e_1 \longmapsto e_1'}{\mathsf{cat}(e_1; e_2) \longmapsto \mathsf{cat}(e_1'; e_2)} \qquad \textsc{D-Cat-2}\ \frac{e_1\ \mathsf{val} \quad e_2 \longmapsto e_2'}{\mathsf{cat}(e_1; e_2) \longmapsto \mathsf{cat}(e_1; e_2')}$$

$$\textsc{D-Len}\ \frac{|s| = n}{\mathsf{len}(\mathsf{str}[s]) \longmapsto \mathsf{num}[n]} \qquad \textsc{D-Len-1}\ \frac{e \longmapsto e'}{\mathsf{len}(e) \longmapsto \mathsf{len}(e')} \qquad \textsc{D-Let}\ \frac{}{\mathsf{let}(e_1; x.\, e_2) \longmapsto e_2[e_1/x]}$$

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

Terms can be thought of as **states** of a transition system. The judgement $e_1 \longmapsto e_2$ 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 ruiles, 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 $\mathsf{plus}(e_1; e_2)$.

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

$$\dfrac{\dfrac{}{\mathsf{len}(\mathsf{str}[\text{‘asdf’}]) \longmapsto \mathsf{num}[4]}\ \textsc{D-Len}}{\mathsf{plus}(\mathsf{len}(\mathsf{str}[\text{‘asdf’}]); \mathsf{num}[1]) \longmapsto \mathsf{plus}(\mathsf{num}[4]; \mathsf{num}[1])}\ \textsc{D-Plus-1}$$

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

$$\mathsf{plus}(\underline{\mathsf{len}(\mathsf{str}[\text{‘asdf’}])}; \mathsf{num}[1]) \longmapsto \mathsf{plus}(\mathsf{num}[4]; \mathsf{num}[1]) \tag{1}$$

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.

$$\mathsf{plus}(\underline{\mathsf{len}(\mathsf{str}[\text{‘asdf’}])}; \mathsf{num}[1]) \longmapsto \underline{\mathsf{plus}(\mathsf{num}[4]; \mathsf{num}[1])} \longmapsto \mathsf{num}[5] \tag{2}$$

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

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

$$\textsc{D-Multi-Refl}\ \dfrac{}{e \longmapsto^* e} \qquad\qquad \textsc{D-Multi-Step}\ \dfrac{e \longmapsto e' \qquad e' \longmapsto^* e''}{e \longmapsto^* e''}$$

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

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

Proposition 2. The rule $\dfrac{e_1 \longmapsto^* e_2 \qquad e_2 \longmapsto^* e_3}{e_1 \longmapsto^* e_3}$ is admissible.

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

$$e = e_0 \longmapsto \ \ldots \ \longmapsto e_n = e'$$

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

$$\mathsf{plus}(\mathsf{len}(\mathsf{str}[\text{‘asdf’}]); \mathsf{num}[1]) \longmapsto^* \mathsf{num}[5]$$

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

$$\mathsf{plus}(\mathsf{len}(\mathsf{str}[\text{‘asdf’}]); \mathsf{num}[1]) \longmapsto \mathsf{num}[5]$$

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 val then there is no e' with $e \longmapsto e'$.

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

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

Proposition 4 (Determinism). If $e \longmapsto e_1$ and $e \longmapsto e_2$ 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

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

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