Small Step Semantics for Cardano

October 17, 2018

Abstract

We define the basis for our version of "small-step semantics" as applied to defining components of the Cardano cryptocurrency.

1 Introduction

In defining Cardano, we are concerned with the means to construct inductive datatypes satisfying some validity conditions. For example, we wish to consider when a sequence of transactions forms a valid *ledger*, or a sequence of blocks forms a valid *chain*.

This document describes the semi-formal methods we use to describe such validity conditions and how they result in the construction of valid states.

2 Preliminaries and Notation

In this section, we define some preliminaries and some notation common to Cardano specifications.

Functions and partial functions We denote the set of functions from A to B as $A \to B$, and a function from A to B as $f \in A \to B$ or $f : A \to B$. We denote the set of partial functions from A to B as $A \mapsto B$, and an equivalent partial function as $g \in A \mapsto B$ or $g : A \mapsto B$. Analogously to Haskell, we use spacing for function application. So $f\ a$ represents the application of a to the function f.

3 Transition Systems

In describing the semantics for a system L we are principally concerned with five things:

States The underlying datatype of our system, whose validity we are required to prove.

Transitions The means by which we might move from one (valid) state to another.

Signals The means by which transitions are triggered.

Rules Formulae describing how to derive valid states or transitions. A rule has an *anteccedent* (sometimes called *premise*) and a *consequent* (sometimes called *conclusion*), such that if the conditions in the antecedent hold, the consequent is assumed to be a valid state or transition.

Environment Sometimes we may implicitly rely on some additional information being present in the system to evaluate the rules. An environment does not have to exist (or, equivalently, we may have an empty environment), but crucially an environment is not modified by transitions.

Definition 3.1 (State transition system)**.** A *state transition system* L is given by a 5-tuple $(S, T, \Sigma, R, \Gamma)$ where:

S is a set of (not necessarily valid) states.

Σ is a set of signals.

Γ is a set of environment values.

T is a set of (not necessarily valid) transitions. We have that

$$T \subseteq \mathbb{P}\left(\Gamma \times S \times \Sigma \times S\right)$$

R is a set of derivation rules. A derivation rule is given by two parts: its antecedent is a logical formula in $S \cup \Sigma \cup \Gamma$. Its consequent is either:

- A state $s \in S$, or
- A transition $t \in T$.

Remark 3.1. The above definition is somewhat redundant, since the transition set T implicitly defines the state, signal and environment sets. We use the above definition for clarity, since we often wish to refer to states and signals directly.

Definition 3.2 (Validity)**.** For a transition system $(S, T, \Sigma, R, \Gamma)$ and environment $\gamma \in \Gamma$, we say that a state $s \in S$ is valid if either:

- It appears as a consequent of a rule $r \in R$ whose antecedent has no variables in $S \cup \Sigma$ and which is true as evaluated at γ, or
- There exists a tuple $(s', \sigma, r, t) \in S \times \Sigma \times R \times T$ such that s' is valid, the antecedent of r is true as evaluated at (γ, s', σ) and where $t = (\gamma, s', \sigma, s)$.

For a state transition system $L = (S, T, \Sigma, R, \Gamma)$, we use the following notation to represent transitions and rules.

Transitions

$$\_ \vdash \_ \xrightarrow[\mathrm{L}]{\_} \_ \in T$$

Rules

$$\text{Rule-name} \frac{\textit{antecedent}}{\textit{consequent}} \in R$$

Figure 1: Notation for a state transition system.

Figure 3 gives the notation for transitions and rules. For notational convenience, we allow for some additional constructions in the antecedent:

- Rather than a single logical predicate in the antecedent, we allow multiple predicate formulae. All predicates defined in the antecedent must be true; that is, the antecedent is treated as the conjunction of all predicates defined there.
- The antecedent may contain variable definitions of the form $z = f(s)$, where z is an additional binding to be introduced. We will refer to these as "let-bindings" in analogy with their use in programming languages.

We can give an example of such a rule:

$$\text{Example}\frac{P(A) \qquad z = f\ s \qquad B = g(A, z)}{E1 \vdash A \xrightarrow[\text{S2}]{s} B}$$

To read such a rule, we proceed as follows:

1. Firstly, we look at the LHS of the consequent. This introduces the basic environment, initial state and signal.
2. We then expand our set of variables using let bindings in the antecedent.
3. We evaluate all predicates in the antecedent ($P(A)$ in the example above), which should now be defined in terms of $S \cup \Sigma \cup \Gamma$ and the variables introduced in step 2.
4. If all predicates evaluate to true, then we may assume a valid transition to the RHS of the consequent.

4 Composition of Transition Systems

Part of the benefit of this approach is that it allows us to define systems which themselves rely on other state transition systems. To do this, we allow for an additional construction in the antecedent.

- The antecedent may contain a transition of the form $ENV1 \vdash A \xrightarrow[\text{SYS}]{s} B$. This may either act as a predicate (if B is already bound) or as an introduction of B as per the state transition rules for system SYS.

We then allow a construction as follows:

$$\text{Example}\frac{P(A) \qquad z = f\ s \qquad E1 \vdash A \xrightarrow[\text{S1}]{z} B}{E1 \vdash A \xrightarrow[\text{S2}]{s} B}$$

The transition in the antecedent acts as a predicate requiring that A transitions to B under the system $S1$, and, assuming B is otherwise free (has not been bound by the LHS of the consequent or by another let binding), acts as an introduction rule for B as determined by the transition rules for system $S1$.

5 Serialisation

Various of our specifications will rely on a serialisation function which encodes a variety of types to a binary format.

$[\![\_]\!] \in \mathbf{Hask} \mapsto \_ \to \mathbf{Bytes}$	serialisation
$[\![\_]\!]_A \in A \to \mathbf{Bytes}$	

Figure 2: Serialisation functions

Figure 5 gives the definition of a serialisation function. We define this as a partial function since it is defined only on a subset of Haskell types. As an abuse of notation, we drop the

subscript A for the serialisation function on type A, since it is unambiguously determined by its first argument.

We require our serialisation function to satisfy the following property:

Definition 5.1 (Distributivity over injective functions)**.** Let $s : \mathcal{C} \mapsto \_ \to D$ be a function defined over a subset of objects in $\mathcal{C}$, such that $s_A : A \to D$ for all $A \in \operatorname{dom} s$. We say that s *distributes over injective functions* if, for all injective functions $f : A \to B$ where $A, B \in \operatorname{dom} s$, there exists a function $f_s : D \to D$ such that the following diagram commutes:

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ \downarrow{\scriptstyle s_A} & & \downarrow{\scriptstyle s_B} \\ D & \xrightarrow{f_s} & D \end{array}$$

This property allows us to extract specific subparts of our serialised form. Our specifications will rely in places on functions defined over the serialisesation of a value. If we have a type $A = (B, C, D, E)$, for example, and we need to extract $[\![B]\!]$ from $[\![A]\!]$, it is this property which guarantees we may do so without rounnd-tripping through A, since $[\![\pi_1\ A]\!] = (\pi_1)_{[\![]\!]}\ [\![A]\!]$ (where π_1 is the projection onto the first component of the tuple).