simpleFUN Semantics

1 The SimpleFUN Abstract Syntax

We include the abstract syntax here for easy reference when studying the domains and transition rules in the following sections. There is one minor change in the abstract syntax from handout 4: we explicitly include expressions using the unary and binary operators ($\neg e$ and $e_1 \oplus e_2$, respectively). They are important in deciding when to create a **notK** or **binopLeftK**/**binopRightK** continuation, respectively (see subsection 2.2). Thus, we have a way of differentiating between a function call that uses a user-defined function ($e_f(e_1 \ldots e_n)$, where e_f is not a unary or binary operator) and one that uses a built-in operator ($\neg e$ and $e_1 \oplus e_2$). Note that $e_1 \oplus e_2$ is shorthand for $\oplus(e_1, e_2)$, and that $\neg e$ is shorthand for $\neg(e)$ (meaning in both cases, they are essentially function calls, where the function is the operator).

$$x \in \mathit{Variable} \qquad n \in \mathbb{N} \qquad b \in \mathit{Bool} \qquad \mathit{name}, \mathit{cons}, \mathit{fld} \in \mathit{Label} \qquad \oplus \in \mathit{BinaryOp}$$

$$\begin{aligned}
\mathit{prog} \in \mathit{Program} &::= \mathit{typedef}_1 \ldots \mathit{typedef}_n \;\; e \\
\mathit{typedef} \in \mathit{TypeDef} &::= \textbf{type}\; \mathit{name} = \mathit{cons}_1 : \tau_1 \ldots \mathit{cons}_n : \tau_n \\
e \in \mathit{Exp} &::= x \mid n \mid b \mid \textbf{nil} \mid \neg e \mid e_1 \oplus e_2 \mid (x_1 : \tau_1 \ldots x_n : \tau_n) \Rightarrow e \mid e_f(e_1 \ldots e_n) \\
&\mid \textbf{if}\; e_1\; e_2\; e_3 \mid \textbf{let}\; x = e_1\; \textbf{in}\; e_2 \mid \textbf{rec}\; x : \tau = e_1\; \textbf{in}\; e_2 \mid [\mathit{fld}_1 = e_1 \ldots \mathit{fld}_n = e_n] \\
&\mid e.\mathit{fld} \mid \mathit{cons}\; e \mid \textbf{case}\; e\; \textbf{of}\; \mathit{cons}_1\; x_1 \Rightarrow e_1 \ldots \mathit{cons}_n\; x_n \Rightarrow e_n
\end{aligned}$$

2 Semantics

In this section we describe the following: (1) the semantic domains; (2) the state transition rules; and (3) the helper functions used by the above descriptions.

2.1 Semantic Domains

The following figure describes SimpleFUN's semantic domains, or the set of things that have and give **meaning** to the language. Evaluating a SimpleFUN expression means to turn a syntactic expression (for example, $1 + (2 * 3)$) into some final semantic value (for example, the natural number 7). When evaluating an expression, it is often necessary to evaluate its subexpressions first. We use a construct called a **continuation** to do so, which allows us to remember the larger expression we are evaluating before we begin evaluating its smaller subexpressions. Upon finishing the evaluation of some expression and returning some semantic value, we can look at the current continuation to see how to use this value in some larger context (explained in more detail below).

You may notice that unlike the abstract syntax, which uses the ::= and | symbols in creating production rules, the figure below uses = to separate domains and + to separate values within each domain. This change is notation is to reflect that we are defining semantic objects, rather than syntactic objects. The vector notation represents an ordered sequence where (by abuse of notation) the sequence is of unspecified length n and the subscript i ranges from 1 to n.

$$\mathcal{F} \in \mathit{TransitionFunction} = \mathit{State} \to \mathit{State}$$

$$\begin{aligned}
\varsigma \in \mathit{State} &= \mathit{Term} \times \mathit{Env} \times \overrightarrow{\mathit{Kont}_i} \\
t \in \mathit{Term} &= \mathit{Exp} + \mathit{Value} \\
\rho \in \mathit{Env} &= \mathit{Variable} \to \mathit{Value} \\
v \in \mathit{Value} &= \mathit{NumV} + \mathit{BoolV} + \mathit{NilV} + \mathit{ClosureV} + \mathit{RecordV} + \mathit{ConstructorV} + \mathit{LetRecV} \\
nv \in \mathit{NumV} &= \textbf{numV}(\mathbb{N}) \\
bv \in \mathit{BoolV} &= \textbf{boolV}(\textbf{true} + \textbf{false}) \\
u \in \mathit{NilV} &= \textbf{nilV} \\
clo \in \mathit{ClosureV} &= \textbf{closureV}(\overrightarrow{\mathit{Variable}_i}, \mathit{Exp}, \mathit{Env}) \\
r \in \mathit{RecordV} &= \textbf{recordV}(\mathit{Label} \to \mathit{Value}) \\
con \in \mathit{ConstructorV} &= \textbf{constructorV}(\mathit{Label}, \mathit{Value}) \\
rl \in \mathit{LetRecV} &= \textbf{letrecV}(\mathit{Variable}, \mathit{Exp}, \mathit{Env}) \\
\kappa \in \mathit{Kont} &= \textbf{binopLeftK}(\oplus, \mathit{Exp}) + \\
&\quad\ \textbf{binopRightK}(\oplus, \mathit{Value}) + \\
&\quad\ \textbf{notK} + \\
&\quad\ \textbf{appK}(\overrightarrow{\mathit{Exp}_i}, \overrightarrow{\mathit{Value}_i}) + \\
&\quad\ \textbf{ifK}(\mathit{Exp}, \mathit{Exp}) + \\
&\quad\ \textbf{letK}(\mathit{Variable}, \mathit{Exp}) + \\
&\quad\ \textbf{recordK}(\overrightarrow{\mathit{Label}_i}, \overrightarrow{\mathit{Exp}_i}, \overrightarrow{\mathit{Value}_i}) + \\
&\quad\ \textbf{accessK}(\mathit{Label}) + \\
&\quad\ \textbf{consK}(\mathit{Label}) + \\
&\quad\ \textbf{caseK}(\overrightarrow{(cons\ x \Rightarrow e)_i}) + \\
&\quad\ \textbf{restoreK}(\mathit{Env})
\end{aligned}$$

2.1.1 Values

A value can be a number, boolean, nil, closure (a function and its environment), record, constructor (user-defined type), and recursive let. A *NumV* represents the semantic value of a number. A *BoolV* represents the semantic value of a boolean. A *ClosureV* contains a function and the environment that was present when it was created (used to capture any of its free variables). *RecordV* and *ConstructorV* represent the semantic values of their syntactic counterparts. A *LetRecV* is used to allow recursion in the language, by giving us a structure that holds the unevaluated recursive expression, and the variable to which it should be bound, until needed.

2.1.2 Continuations

As introduced above, a **continuation** is a construct used to model control flow in this purely functional language, meaning we can explicitly and deterministically control which expression we need to currently evaluate as well as remember any expressions we need to evaluate afterwards.

Continuation Example For example, say we have some SimpleFUN expression like $(3+4)+(5*6)$. To evaluate it, we will have to perform an addition between the operands (3 + 4) and (5 ∗ 6). However, neither of these operands are values yet, so they too must be further evaluated. We say that the next expression to evaluate is (3 + 4), and we *remember* that we were in the middle of trying to add the expression (3 + 4) to the expression (5 ∗ 6). This act of remembering is creating a continuation (here, a **binopLeftK**) and putting said continuation on our continuation stack. Then we proceed with the evaluation of the subexpression (3 + 4). As explained above, we differentiate between the *syntactic* number 3, which is an expression, and the *semantic* number, or actual value, of 3. Because of this, we need to evaluate each of the operands in this subexpression as well. To do that, we make the expression 3 the current expression under evaluation, remembering that we were in the middle of trying to add the expression 3 to the expression 4 by creating *another* continuation (a **binopLeftK**) and putting it on top of the continuation stack. Now that the current expression under evaluation is 3, we know to convert that syntactic 3 into a semantic 3 value (here there is no tangible difference between the syntactic 3 and the semantic 3 (i.e. its mathematical meaning); however, this will not always be the case, so we must use an explicit rule that shows the correspondence between syntax and mathematical meaning, even for numbers).

Now that we have a value, we look on top of the continuation stack to figure out what we were doing beforehand that necessitated getting this value 3. Our top continuation is a **binopLeftK**, which tells us that we were in the middle of adding two operands and that the current value we have is the *left* operand. We then proceed to make the right operand expression, 4, the current expression under evaluation, remembering that we were in the middle of an addition operation *and* that we already know the left operand, which is the value 3. To remember this, we pop off the old **binopLeftK** continuation and put a **binopRightK** continuation on the continuation stack. Then we proceed evaluating the expression 4, which becomes the value 4, and again we look on top of the continuation stack to see what we were in the middle of doing. Since the top continuation is a **binopRightK**, we know that the current value 4 is the *right* operand of the addition, and since we now have evaluated both operands, we can add them together to get the value 7. We pop off the old **binopRightK** continuation and see that the top of the continuation stack is **binopLeftK**; this is because we originally put this continuation on the stack when we began evaluating the expression (3 + 4). Now that we know that (3 + 4) evaluates to 7 and that the top continuation is a **binopLeftK**, we know that we were in the middle of an addition operation and that the left operand is 7. We proceed by popping off this **binopLeftK** continuation, setting the right-hand side expression (5 ∗ 6) as the current expression under evaluation, and placing a **binopRightK** on the continuation stack. After following the steps as we did previously, except now for the expression (5 ∗ 6), we'll eventually get to the point that the current value we just evaluated is 30 and the top continuation is **binopRightK**. Since both operands 7 and 30 are fully evaluated, we pop off that **binopRightK** continuation from the stack and see that we can finally do that original addition we wanted to do all along, resulting in the value 37. Since we just evaluated a value, and since the continuation stack will be empty at this point (meaning there's nothing else to do), we know we have finished evaluation of the entire program.

Continuation Descriptions The following are descriptions of each of the continuations that can be placed on the continuation stack $\vec{\kappa}$. These descriptions are a high-level overview of the basic way each is used. To get a clearer picture of exactly how they are created and used, please study the transition rules in subsection 2.2, especially noting how the $\overrightarrow{\kappa_{new}}$ column changes during the evaluation of most of the expressions in rules #1 through #15. During rules #16 through #32, which show how to handle terms which are values, the continuation stack is often inspected as part of the premises; the continuation on top of the stack tells us what expression or operation we were in the middle of evaluating and which term we need to look at next.

binopLeftK A **binopLeftK** continuation is used to remember 1) that the current expression under evaluation is the left-hand operand of some binary operation, 2) the binary operator, and 3) the right-hand operand to be evaluated later. When we eventually evaluate the left-hand operand down to a single value and see that the continuation on the top of the stack is a **binopLeftK**, we know that we must now begin evaluating the right-hand operand, which was saved for us in the **binopLeftK**.

binopRightK A **binopRightK** continuation is used to remember that 1) that the current expression under evaluation is the right-hand operand of some binary operation, 2) the binary operator, and 3) the left-hand operand, which was evaluated to a value previously. When we eventually evaluate the right-hand operand down to a single value and see that the continuation on top of the stack is **binopRightK**, we know that we have evaluated both operands and can now perform the actual operation with that saved operator and the two operand values.

notK A **notK** continuation is used to remember that the current expression under evaluation needs to be negated when it finally becomes a value. So when we eventually evaluate the expression down to a value, assuming that value is a boolean and the continuation on top of the stack is a **notK**, we know that we must produce the negated version of the current boolean value.

appK An **appK** continuation is used to remember that we are in the middle of evaluating the function and arguments to a function call. When we encounter a function call expression, we evaluate the function expression and use an **appK** continuation to store the list of unevaluated argument expressions and the list of evaluated arguments and the eventual function value. Whenever we have evaluated an expression down to a value *and* the top of the continuation stack is an **appK**, we add the current value to that list of evaluated arguments and set the current expression under evaluation to be the head of the list of unevaluated argument expressions. When we finally reach a value *and* the list of unevaluated expressions in the **appK** continuation on the top of the stack is empty, we know that we can proceed with the function call (meaning we set the new expression under evaluation to be the function's body, updating the environment to include a mapping from the function's formal paramters to the actual evaluated argument values).

ifK An **ifK** continuation is used to remember that we're in the middle of evaluating an **if** expression. When we first encounter an **if** expression, we set the condition expression as the expression under evaluation and create an **ifK** continuation to remember the expressions to evaluate if the condition is **true** and if it is **false**. When we finally evaluate the condition expression to a value and see that the top of the continuation stack holds an **if** continuation, we set the new expression under evaluation to be one of the two expressions saved in the **if** continuation, depending on the whether the value of the condition was **true** or **false**.

letK A **letK** continuation is used to remember that the current expression under evaluation needs to be bound to the variable being introduced in the **let** expression when the current expression eventually evaluates to a value, so that we can then begin evaluating the right-hand side of the **let** expression with this new variable-to-value mapping added to the environment.

recordK A **recordK** continuation is used to remember 1) that we are in the middle of evaluating a record expression and 2) the list of fields that have been evaluated and remain to be evaluated. This is so that when we reach a value and see that the top of the continuation stack is a **recordK** continuation, we know to add the current value to the list of saved values and then continue evaluation by evaluating one of the remaining unevaluated expressions used in constructing the record. When the **recordK** continuation's list of unevaluated expressions is empty, we can finally proceed by creating a record value filled with the list of values we had been saving all along.

accessK A **accessK** continuation is used to remember 1) that the current expression under evaluation should be a record when finally evaluated and 2) the field which we wish to extract out of this record value.

consK A **consK** continuation is used to remember 1) the name of a constructor being created and 2) that the current expression under evaluation, when finally evaluated, should be the value used to create the constructor value.

caseK A **caseK** continuation is used to remember 1) that the current expression under evaluation should eventually be a user-defined constructor to be matched against and 2) the list of branches to take. The branch to take will be the branch whose constructor name matches the eventual value of the expression under evaluation.

restoreK A **restoreK** continuation is used to remember some environment before evaluating expressions that may alter the current environment, so that upon reaching a state where the top of the continuation stack is a **restoreK**, we may continue evaluation with the original environment.

2.2 State Transition Rules

The transition function $\mathcal{F}$ is defined in the table below. We treat the sequence of continuations $\vec{\kappa}$ as a stack, where κ is the top of the continuation stack in the source state and $\vec{\kappa_1}$ is the rest of that continuation stack. We use $::$ to indicate the concatenation of sequences, so that $\vec{\kappa} = \kappa :: \vec{\kappa_1}$. For example, when we write **appK**$(\vec{e_i}, v :: \vec{v_i}) :: \vec{\kappa_1}$, this is the same as if we had popped the top continuation κ off the stack and then pushed an **appK**$(\vec{e_i}, v :: \vec{v_i})$ continuation onto the remaining part of the stack $\vec{\kappa_1}$ like so:

$$\frac{\textbf{appK}(\vec{e_i}, v :: \vec{v_i})}{\vec{\kappa_1}}$$

This corresponds to the manipulation of the continuation stack that occurs in rule #23.

A Comment on Notation In the table of transition rules below, we rely on special notation to be both precise and concise. We use "_" when we don't care about the value in a particular position. We use $\rho[x \mapsto v]$ to indicate updating the environment ρ to include a mapping from the key x to the value v, yielding a new environment in the process. Similarly, we use $\rho[\vec{x_i} \mapsto \vec{v_i}]$ to indicate updating the environment ρ to include a mapping from a sequence of keys $\vec{x_i}$ to a sequence of values $\vec{v_i}$; thus, it is shorthand for $\rho[x_1 \mapsto v_1, x_2 \mapsto v_2, \ldots, x_{n-1} \mapsto v_{n-1}, x_n \mapsto v_n]$. Implicit in this notation is that $|\vec{x_i}| = |\vec{v_i}| = n$.

Table 1: The transition function $\mathcal{F}$. This moves from an input state $(\langle t, \rho, \vec{\kappa} \rangle)$ to the next state $(\langle t_{new}, \rho_{new}, \overrightarrow{\kappa_{new}} \rangle)$. A program terminates normally when our term t is a non-**letrecV** value and the continuation stack $\vec{\kappa}$ is empty; we represent this by trivially looping forever via rule #17.

#	t	Premises	t_{new}	ρ_{new}	$\overrightarrow{\kappa_{new}}$
1	x	$x \in \texttt{keys}(\rho), v = \rho(x)$	v	ρ	$\vec{\kappa}$
2	n		$\textbf{numV}(n)$	ρ	$\vec{\kappa}$
3	b		$\textbf{boolV}(b)$	ρ	$\vec{\kappa}$
4	**nil**		**nilV**	ρ	$\vec{\kappa}$
5	$e_1 \oplus e_2$		e_1	ρ	$\textbf{binopLeftK}(\oplus, e_2) :: \vec{\kappa}$
6	$\neg e$		e	ρ	$\textbf{notK} :: \vec{\kappa}$
7	$(x_1 : \tau_1 \ldots x_n : \tau_n) \Rightarrow e$		$\textbf{closureV}(x_1 \ldots x_n, e, \rho)$	ρ	$\vec{\kappa}$
8	$e_f(e_1 \ldots e_n)$		e_f	ρ	$\textbf{appK}(e_1 \ldots e_n, []) :: \vec{\kappa}$
9	**if** e_1 e_2 e_3		e_1	ρ	$\textbf{ifK}(e_2, e_3) :: \vec{\kappa}$
10	**let** $x = e_1$ **in** e_2		e_1	ρ	$\textbf{letK}(x, e_2) :: \vec{\kappa}$
11	**rec** $x : \tau = e_1$ **in** e_2		e_2	$\rho[x \mapsto \textbf{letrecV}(x, e_1, \rho)]$	$\textbf{restoreK}(\rho) :: \vec{\kappa}$
12	$[\mathit{fld}_1 = e_1 \ldots \mathit{fld}_n = e_n]$		e_1	ρ	$\textbf{recordK}(\mathit{fld}_1 \ldots \mathit{fld}_n, e_2 \ldots e_n, []) :: \vec{\kappa}$
13	$e.\mathit{fld}$		e	ρ	$\textbf{accessK}(\mathit{fld}) :: \vec{\kappa}$
14	$\mathit{cons}\ e$		e	ρ	$\textbf{consK}(\mathit{cons}) :: \vec{\kappa}$
15	**case** e **of** $\overrightarrow{(\mathit{cons}\ x \Rightarrow e)}_i$		e	ρ	$\textbf{caseK}(\overrightarrow{(\mathit{cons}\ x \Rightarrow e)}_i) :: \vec{\kappa}$
16	$\textbf{letrecV}(x, e_1, \rho')$		e_1	$\rho'[x \mapsto \textbf{letrecV}(x, e_1, \rho')]$	$\textbf{restoreK}(\rho) :: \vec{\kappa}$
17	v	$v \neq \textbf{letrecV}(\_, \_, \_), \vec{\kappa} = []$	v	ρ	$[]$
18	v	$v \neq \textbf{letrecV}(\_, \_, \_), \kappa = \textbf{restoreK}(\rho')$	v	ρ'	$\vec{\kappa_1}$
19	v	$v \neq \textbf{letrecV}(\_, \_, \_), k = \textbf{binopLeftK}(\oplus, e)$	e	ρ	$\textbf{binopRightK}(\oplus, v) :: \vec{\kappa_1}$
20	v_2	$v_2 \neq \textbf{letrecV}(\_, \_, \_), k = \textbf{binopRightK}(\oplus, v_1)$, $v = \texttt{valueOf}(\oplus, v_1, v_2)$	v	ρ	$\vec{\kappa_1}$
21	$\textbf{boolV}(\textbf{true})$	$k = \textbf{notK}$	$\textbf{boolV}(\textbf{false})$	ρ	$\vec{\kappa_1}$
22	$\textbf{boolV}(\textbf{false})$	$k = \textbf{notK}$	$\textbf{boolV}(\textbf{true})$	ρ	$\vec{\kappa_1}$
23	v	$v \neq \textbf{letrecV}(\_, \_, \_), k = \textbf{appK}(e :: \vec{e_i}, \vec{v_i})$	e	ρ	$\textbf{appK}(\vec{e_i}, v :: \vec{v_i}) :: \vec{\kappa_1}$
24	v	$v \neq \textbf{letrecV}(\_, \_, \_), k = \textbf{appK}([], \vec{v_i})$ $v_f :: \overrightarrow{v_{args}} = \texttt{reverse}(v :: \vec{v_i})$, $v_f = \textbf{closureV}(\vec{x_i}, e, \rho'), \lvert\overrightarrow{v_{args}}\rvert = \lvert\vec{x_i}\rvert$	e	$\rho'[\vec{x_i} \mapsto \overrightarrow{v_{args}}]$	$\textbf{restoreK}(\rho) :: \vec{\kappa_1}$
25	$\textbf{boolV}(\textbf{true})$	$k = \textbf{ifK}(e_2, e_3)$	e_2	ρ	$\vec{\kappa_1}$
26	$\textbf{boolV}(\textbf{false})$	$k = \textbf{ifK}(e_2, e_3)$	e_3	ρ	$\vec{\kappa_1}$
27	v	$v \neq \textbf{letrecV}(\_, \_, \_), k = \textbf{letK}(x, e_2)$	e_2	$\rho[x \mapsto v]$	$\textbf{restoreK}(\rho) :: \vec{\kappa_1}$
28	v	$v \neq \textbf{letrecV}(\_, \_, \_), k = \textbf{recordK}(\overrightarrow{\mathit{fld}}_i, e :: \vec{e_i}, \vec{v_i})$	e	ρ	$\textbf{recordK}(\overrightarrow{\mathit{fld}}_i, \vec{e_i}, v :: \vec{v_i}) :: \vec{\kappa_1}$
29	v	$v \neq \textbf{letrecV}(\_, \_, \_), k = \textbf{recordK}(\overrightarrow{\mathit{fld}}_i, [], \vec{v_i})$ $\vec{v}_{all} = \texttt{reverse}(v :: \vec{v_i})$	$\textbf{recordV}([\overrightarrow{\mathit{fld}}_i \mapsto \overrightarrow{v_{all}}])$	ρ	$\vec{\kappa_1}$
30	$\textbf{recordV}([\ldots, \mathit{fld} = v, \ \ldots])$	$k = \textbf{accessK}(\mathit{fld})$	v	ρ	$\vec{\kappa_1}$
31	v	$v \neq \textbf{letrecV}(\_, \_, \_), k = \textbf{consK}(\mathit{cons})$	$\textbf{constructorV}(\mathit{cons}, v)$	ρ	$\vec{\kappa_1}$
32	$\textbf{constructorV}(\mathit{cons}, v)$	$k = \textbf{caseK}(\overrightarrow{(\mathit{cons}\ x \Rightarrow e)}_i)$ $(\mathit{cons}\ x \Rightarrow e) \in \overrightarrow{(\mathit{cons}\ x \Rightarrow e)}_i$	e	$\rho[x \mapsto v]$	$\textbf{restoreK}(\rho) :: \vec{\kappa_1}$

2.3 Helper Functions

We define the helper functions used by the previous sections. The functions are listed in alphabetical order.

2.3.1 `valueOf`

`valueOf` takes a binary operator and two values and returns the result of operating on those two values with said operator. By using the ˆ symbol over an operator, we differentiate between a syntactic operator (e.g. `+`, which is just a token from the parser) and the mathemtical meaning of this operator (e.g. $\hat{+}$, meaning addition). Normally, the semantic meaning of a syntactic operator is obvious, but we must be explicit when defining semantics.

$\texttt{valueOf} \in \mathit{BinaryOp} \times \mathit{Value} \times \mathit{Value} \to \mathit{Value}$

$\texttt{valueOf}(+, \textbf{numV}(n_1), \textbf{numV}(n_2)) = \textbf{numV}(n_1 \hat{+} n_2)$

$\texttt{valueOf}(-, \textbf{numV}(n_1), \textbf{numV}(n_2)) = \textbf{numV}(n_1 \hat{-} n_2)$

$\texttt{valueOf}(\times, \textbf{numV}(n_1), \textbf{numV}(n_2)) = \textbf{numV}(n_1 \hat{\times} n_2)$

$\texttt{valueOf}(\div, \textbf{numV}(n_1), \textbf{numV}(n_2)) = \textbf{numV}(n_1 \hat{\div} n_2)$

$\texttt{valueOf}(<, \textbf{numV}(n_1), \textbf{numV}(n_2)) = \textbf{boolV}(n_1 \hat{<} n_2)$

$\texttt{valueOf}(=, \textbf{numV}(n_1), \textbf{numV}(n_2)) = \textbf{boolV}(n_1 \hat{=} n_2)$

$\texttt{valueOf}(\wedge, \textbf{boolV}(b_1), \textbf{boolV}(b_2)) = \textbf{boolV}(b_1 \hat{\wedge} b_2)$

$\texttt{valueOf}(\vee, \textbf{boolV}(b_1), \textbf{boolV}(b_2)) = \textbf{boolV}(b_1 \hat{\vee} b_2)$

2.3.2 `reverse`

`reverse` reverses the elements of the given list. Because its implementation is standard, we do not bother to list it explicitly.

$\texttt{reverse} \in \mathit{List}[A] \to \mathit{List}[A]$

3 Example

Example Program `call1`:
((x: num) ⇒x)(7)

Table 2: The sequence of state transitions for program `call1`.

Rule #	t	ρ	$\vec{\kappa}$
8	((x : num) ⇒ x)(7)	{}	[]
7	((x : num) ⇒ x)	{}	**appK**([7], []) :: []
23	**closureV**([x], x, {})	{}	**appK**([7], []) :: []
2	7	{}	**appK**([], [**closureV**([x], x, {})]) :: []
24	**numV**(7)	{}	**appK**([], [**closureV**([x], x, {})]) :: []
1	x	{x ↦ **numV**(7)}	**restoreK**({}) :: []
18	**numV**(7)	{x ↦ **numV**(7)}	**restoreK**({}) :: []
17	**numV**(7)	{}	[]

4 Getting Stuck

In general, it is not always possible to transition from one state to another state. For example, consider the following program:

1 + **true**

Eventually, this program will reach a point where execution cannot proceed. Specifically, this will occur upon a call of the `valueOf` helper function, with the + operation. The + operation expects both of its arguments to be instances of *NumV*, but in this case + is being asked to operate on an instance of *NumV* and an instance of *BoolV*. There is no listed behavior for this case, so execution cannot proceed. This is referred to as "getting stuck". If we were to try to draw out the automata for this program's execution, it would abruptly stop at the state just before the fateful call to the `valueOf` helper function.

4.1 Relationship to Types

Oftentimes, stuck programs are somehow inherently erroneous. Stuck programs, by their construction, attempt to do something for which we intentionally did not specify a behavior for, as no behavior made sense. Understandably, programmers usually do not want their programs to get stuck, as stuck programs do not usually do useful work.

Type systems are commonly employed to help avoid stuck programs. Observe that with our example, the typechecker you previously wrote for `simpleFUN` would **reject** this program ahead of time. In this way, static type systems try to eliminate programs which we know may get stuck at runtime.

An alternative to rejecting programs which may get stuck ahead of time is to modify the semantics themselves to never get stuck. With the previous example, this would mean adding a catch-all case to `valueOf` of some sort, describing exactly what to do if we receive a nonsensical input. A simple approach is to return some sort of error code or to otherwise indicate that an error has occurred, ideally in a way that the program itself can respond to the error. Importantly, these errors exist *in the semantics themselves*. This approach is often employed in dynamically-typed languages upon encountering a runtime type error. Even statically-typed languages usually have to do at least some runtime checks to avoid getting stuck, such as bounds checking for array accesses in Java.

While programmers generally want to know when their programs get stuck (as this usually indicates some sort of programming error), observe that this comes at a cost: static type checking can be both difficult to perform and restrictive, and dynamically checking types incurs runtime overhead. As such, some language implementations effectively give up on informing programmers that their programs get stuck. This can simplify the implementation significantly without sacrificing performance or flexibility, though at the cost of making the language less safe and reliable. This approach is taken by *weakly typed* languages like C, which give up on trying to catch a number of programming errors, such as accessing out-of-bounds elements in arrays. Instead, C puts this burden on programmers themselves, who "promise" the C compiler that they will not write programs that can ever possibly get stuck. If this promise is ever broken, the behavior of the C code becomes *undefined*, meaning **anything** can happen. From an automata perspective, undefined behavior means that instead of abruptly ending the automata at a given state S, S will instead transition to some **completely arbitrary** state S'. Moreover, based on the definitions specified in the C standard, the transition rules themselves get **thrown out** for all subsequent execution: S' may transition to another completely arbitrary state, and there is no longer any way to predict what the automata will do. In practice, this leads to all sorts of nasty bugs, and has been the basis for major exploits like Heartbleed and the recent Cloudflare leaks.