Assignment #5, Update 5: Abstract Machines and Parallelism

15-312: Principles of Programming Languages

Out: Tuesday, April 8, 2014
Due: Thursday, April 24, 2014 1:29PM

Introduction

In this assignment, we will develop a language with the parallel programming primitive of *sequences* and we will write programs in that language. The work will be divided into two stages. In the first stage, we will write a *control stack machine* that controls the evaluation of the sequential portion of the language. In the second stage, we will extend this to implement the parallel constructs of the language.

Submission

We will collect *exactly* the following files from the `/afs/andrew/course/15/312/` directory:

```
handin/<yourandrewid>/assn5/assn5.pdf
handin/<yourandrewid>/assn5/dynamics.sml
handin/<yourandrewid>/assn5/scheduler.sml
handin/<yourandrewid>/assn5/mergesort.par
```

Make sure that your files have the right names (especially `assn5.pdf`!) and are in the correct directories.

1 A Data Parallel Language

The language of study in this assignment features a new type, $\texttt{seq}(\tau)$, for sequences containing elements of type τ. When we create a new sequence, the contents of the resulting array are evaluated in parallel. This effectively allows the programmer to spawn an unbounded number of parallel tasks simultaneously. The remaining elements of the language are sequentially evaluated in left-to-right order.

Types and expressions are given in Figure 1. This language departs a bit from the familiar PCF language, introducing a number of integer operations as primitives and defining operations that work on those values. There are four operations for manipulating sequences:

- $\texttt{tab}[\tau](x.e; e_n)$ – If e_n evaluates to $\texttt{num}[n]$ where $n > 0$, this expression creates a new sequence of length e_n by evaluating $[\overline{0}/x]e, \ldots, [\overline{n-1}/x]e$.
- $\texttt{sub}(e; e_n)$ – If e_n evaluates to $\bar{i}$ and e evaluates to $\texttt{seq}(e_0, \ldots, e_{n-1})$, this expression evaluates to e_i.
- $\texttt{len}(e)$ – Evaluates to the length of the sequence that e evaluates to.

Sort			Abstract Form	Concrete Form
Type	τ	::=	`int`	`int`
			`bool`	`bool`
			`parr`$(\tau_1;\tau_2)$	$\tau_1 \rightharpoonup \tau_2$
			`prod`$(\tau_1;\tau_2)$	$\tau_1 \times \tau_2$
			`seq`(τ)	`seq`(τ)
Comparisons	*cmp*	::=	`eq`	$=$
			`neq`	$\neq$
			`leq`	$\leq$
			`geq`	$\geq$
Integer ops	*op*	::=	`plus`	$+$
			`minus`	$-$
			`times`	$\times$
			`div`	$\div$
			`mod`	$\%$
Exp	e	::=	x	x
			`let`$(e_0; x.e_1)$	`let` $x = e_0$ `in` e_1
			`lam`$[\tau](x.e)$	`fn` $(x{:}\tau)\ e$
			`ap`$(e_1; e_2)$	$e_1\ (e_2)$
			`fix`$[\tau](x.e)$	`fix` $x{:}\tau$ `is` e
			`pair`$(e_1; e_2)$	$\langle e_1, e_2\rangle$
			`pr`$[\mathtt{l}](e)$	$e \cdot \mathtt{l}$
			`pr`$[\mathtt{r}](e)$	$e \cdot \mathtt{r}$
			`num`$[n]$	$\overline{n}$
			`cmp`$[cmp](e_1; e_2)$	$e_1\ cmp\ e_2$
			`calc`$[op](e_1; e_2)$	$e_1\ op\ e_2$
			`true`	`true`
			`false`	`false`
			`if`$(e; e_t; e_f)$	`if` e `then` e_t `else` e_f
			`seq`$(e_0, \ldots, e_{n-1})$	`seq`$(e_0, \ldots, e_{n-1})$
			`tab`$[\tau](x.e; e_n)$	$[e \mid x < e_n]$
			`sub`$(e; e_n)$	$e[e_n]$
			`len`(e)	`len`(e)
			`showt`$(e; x.e_1; y_l.y_r.e_2)$	`showt` e $\{$`elt` $x \Rightarrow e_1 \mid$ `node` $y_l\ y_r \Rightarrow e_2\}$

Figure 1: Expressions and types in our parallel language

- $\texttt{showt}(e; x.e_1; y_l.y_r.e_2)$ – If e evaluates to a sequence with one element v, then this evaluates $[v/x]e_1$. If e evaluates to a sequence with more than one element, then the sequence gets split into two equal sized arrays, the first half-sequence (which may be one element smaller than the second half-sequence) is substituted for y_l in e_2, and the second half-sequence is substituted for y_r in e_2.

To help you test your sequential implementation, we will allow you to explicitly write sequences in the dynamic semantics, but the statics of our language will require that sequences start out containing only values (more on this in a moment). Therefore, it would be better to think of them as part of the internal language, not available to the programmer. This is essentially how we thought of $\texttt{num}[n]$ in Dynamic PCF.

1.1 Values

There's nothing particularly surprising about the values for this language, except that we're using lazy pairs this time around:

$$\frac{}{\texttt{lam}[\tau](x.e)\ \mathsf{val}}(\texttt{lam-val}) \qquad \frac{}{\texttt{pair}(e_1; e_2)\ \mathsf{val}}(\texttt{pair-val}) \qquad \frac{}{\texttt{num}[n]\ \mathsf{val}}(\texttt{num-val})$$

$$\frac{}{\texttt{true}\ \mathsf{val}}(\texttt{true-val}) \qquad \frac{}{\texttt{false}\ \mathsf{val}}(\texttt{false-val}) \qquad \frac{e_0\ \mathsf{val} \quad \ldots \quad e_{n-1}\ \mathsf{val}}{\texttt{seq}(e_0, \ldots, e_{n-1})\ \mathsf{val}}(\texttt{seq-val})$$

1.2 Statics

The static semantics for variables x, $\texttt{let}(e_0; x.e_1)$, $\texttt{lam}[\tau](x.e)$, $\texttt{ap}(e_1; e_2)$, and $\texttt{fix}[\tau](x.e)$, $\texttt{pair}(e_1; e_2)$, $\texttt{pr}[\texttt{l}](e)$, and $\texttt{pr}[\texttt{r}](e)$ remain the same as they have been in previous assignments, and in Homework 2 in particular. Rules for numbers and booleans are straightforward:

$$\frac{}{\Gamma \vdash \texttt{num}[n] : \texttt{int}}(\texttt{int-}I) \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{cmp}[\mathit{cmp}](e_1; e_2) : \texttt{bool}}(\texttt{int-}E_1)$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{calc}[\mathit{op}](e_1; e_2) : \texttt{int}}(\texttt{int-}E_2) \qquad \frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}(\texttt{bool-}I_1) \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}(\texttt{bool-}I_1)$$

$$\frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma \vdash e_t : \tau \qquad \Gamma \vdash e_f : \tau}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_t\ \texttt{else}\ e_f : \tau}(\texttt{bool-}E)$$

By this point in the semester, the following rules for dealing with sequences will hopefully also be straightforward, with the exception that the value judgment is now being referenced as part of the *statics* in rule $\texttt{seq-}I_1$. In fact, in the abstract machine dynamics we give in Section 2, we won't mention the value judgment at all: it moves completely to the statics in the abstract machine presentation used in this assignment.

$$\frac{\Gamma \vdash e_0 : \tau \qquad e_0 \ \mathsf{val} \qquad \ldots \qquad \Gamma \vdash e_{n-1} : \tau \qquad e_{n-1} \ \mathsf{val}}{\Gamma \vdash \mathtt{seq}(e_0, \ldots, e_{n-1}) : \mathtt{seq}(\tau)}(\mathtt{seq}\text{-}I_1)$$

$$\frac{\Gamma, x : \mathtt{int} \vdash e : \tau \qquad \Gamma \vdash e_n : \mathtt{int}}{\Gamma \vdash \mathtt{tab}[\tau](x.e; e_n) : \mathtt{seq}(\tau)}(\mathtt{seq}\text{-}I_2)$$

$$\frac{\Gamma \vdash e : \mathtt{seq}(\tau) \qquad \Gamma \vdash e_n : \mathtt{int}}{\Gamma \vdash \mathtt{sub}(e; e_n) : \tau}(\mathtt{seq}\text{-}E_1) \qquad \frac{\Gamma \vdash e : \mathtt{seq}(\tau)}{\Gamma \vdash \mathtt{len}(e) : \mathtt{int}}(\mathtt{seq}\text{-}E_2)$$

$$\frac{\Gamma \vdash e : \mathtt{seq}(\tau) \qquad \Gamma, x : \tau \vdash e_1 : \tau' \qquad \Gamma, y_1 : \mathtt{seq}(\tau), y_2 : \mathtt{seq}(\tau) \vdash e_2 : \tau'}{\Gamma \vdash \mathtt{showt}(e; x.e_1; y_1.y_2.e_2) : \tau'}(\mathtt{seq}\text{-}E_3)$$

2 Abstract Machines

Abstract machines give a picture of execution that is closer to a realistic implementation than the structural dynamics that have prevailed so far in this course.[1] We are using abstract machines in this assignment because they give us a convenient way of thinking about pausing a sequential process of evaluation, removing a subterm in order to evaluate that subterm in parallel, and then resuming the evaluation of the original term.

Abstract machines record the path from an outermost term to a subterm as a *stack* k, which is made up of a list of *frames* f.

$$\begin{array}{rcl} k & ::= & \epsilon \mid k; f \\ f & ::= & \mathtt{let}(\square; x.e_1) \mid \mathtt{ap}(\square; e_2) \mid \mathtt{ap}(v_1; \square) \mid \mathtt{pr}[\mathtt{l}](\square) \mid \mathtt{pr}[\mathtt{r}](\square) \mid \\ & & \mathtt{cmp}[\mathit{cmp}](\square; e_2) \mid \mathtt{cmp}[\mathit{cmp}](v_1; \square) \mid \mathtt{calc}[\mathit{op}](\square; e_2) \mid \mathtt{calc}[\mathit{op}](v_1; \square) \mid \mathtt{if}(\square; e_t; e_f) \mid \\ & & \mathtt{tab}[\tau](x.e; \square) \mid \mathtt{sub}(\square; e_n) \mid \mathtt{sub}(v; \square) \mid \mathtt{len}(\square) \mid \mathtt{showt}(\square; x.e_1; y_1.y_2.e_2) \end{array}$$

There are three states of the abstract machine: a state s can be either an expression evaluating on top of a stack ($k \rhd e$), a value returning to a stack ($k \lhd v$), or a failure state ($k \blacktriangleleft$). We'll add a fourth, (k $\mathsf{awaiting}[a_0, \ldots, a_{n-1}]$), in the next section.

2.1 Dynamics

The transition relation is an inductive definition, the same as we've seen before for structural dynamics or evaluation dynamics. Presented as an inductive definition, all the rules would look like this:

$$\frac{}{k \rhd \mathtt{let}(e_0; x.e_1) \ \mapsto \ k; \mathtt{let}(\square; x.e_1) \rhd e_0}(\mathtt{let}\text{-}S_1)$$

$$\frac{0 \leq i < n}{k; \mathtt{sub}(\mathtt{seq}(v_0, \ldots, v_{n-1}); \square) \lhd \mathtt{num}[i] \ \mapsto \ k \lhd v_i}(\mathtt{sub}\text{-}S_3)$$

[1] Do remember, though, that this is still a *semantics*, a specification of how implementations should behave and not a guide to implementation. There are elements to this presentation, like the presence of substitution, that you wouldn't want in a realistic implementation. Additionally, the type annotation on the tabulate command $\mathtt{tab}[\tau](x.e; e_n)$ is present in our semantics but is not present in our SML implementation, where we will write `Exp.Tab((x,e),en)`. As with the type annotation on $\mathtt{dcl}[\tau](e; a.e')$ in the last assignment

This would take up a lot of space, and because almost none of the rules of the dynamics have any premises, we'll instead present the dynamics in a compressed format, writing premises off to the side where applicable.

Core (PCF-like) language

$$
\begin{array}{lll}
(\texttt{let}\text{-}S_1) & k \rhd \texttt{let}(e_0; x.e_1) & \mapsto \quad k; \texttt{let}(\square; x.e_1) \rhd e_0 \\
(\texttt{let}\text{-}S_2) & k; \texttt{let}(\square; x.e_1) \lhd v_0 & \mapsto \quad k \rhd [v_0/x]e_1 \\
(\texttt{lam}\text{-}S) & k \rhd \texttt{lam}[\tau](x.e) & \mapsto \quad k \lhd \texttt{lam}[\tau](x.e) \\
(\texttt{ap}\text{-}S_1) & k \rhd \texttt{ap}(e_1; e_2) & \mapsto \quad k; \texttt{ap}(\square; e_2) \rhd e_1 \\
(\texttt{ap}\text{-}S_2) & k; \texttt{ap}(\square; e_2) \lhd v_1 & \mapsto \quad k; \texttt{ap}(v_1; \square) \rhd e_2 \\
(\texttt{ap}\text{-}S_3) & k; \texttt{ap}((\texttt{fn}\,(x{:}\tau)\,e_0); \square) \lhd v_2 & \mapsto \quad k \rhd [v_2/x]e_0 \\
(\texttt{fix}\text{-}S) & k \rhd \texttt{fix}[\tau](x.e) & \mapsto \quad k \rhd [\texttt{fix}[\tau](x.e)/x]e \\
(\texttt{pair}\text{-}S) & k \rhd \texttt{pair}(e_1; e_2) & \mapsto \quad k \lhd \texttt{pair}(e_1; e_2) \\
(\texttt{prl}\text{-}S_1) & k \rhd \texttt{pr}[\texttt{l}](e) & \mapsto \quad k; \texttt{pr}[\texttt{l}](\square) \rhd e \\
(\texttt{prl}\text{-}S_2) & k; \texttt{pr}[\texttt{l}](\square) \lhd \langle e_1, e_2 \rangle & \mapsto \quad k \rhd e_1 \\
(\texttt{prr}\text{-}S_1) & k \rhd \texttt{pr}[\texttt{r}](e) & \mapsto \quad k; \texttt{pr}[\texttt{r}](\square) \rhd e \\
(\texttt{prr}\text{-}S_2) & k; \texttt{pr}[\texttt{r}](\square) \lhd \langle e_1, e_2 \rangle & \mapsto \quad k \rhd e_2
\end{array}
$$

Integers and Booleans When we write ($k \lhd \texttt{num}[n_1\ \mathit{op}\ n_2]$) or ($k \lhd n_1\ \mathit{cmp}\ n_2$), we mean to perform the actual primitive operation on numbers. For example, the state ($k; \texttt{calc}[\texttt{times}](\texttt{num}[3]; \square) \lhd \texttt{num}[4]$) will step to ($k \lhd \texttt{num}[12]$) and the state ($k; \texttt{cmp}[\texttt{neq}](\texttt{num}[3]; \square) \lhd \texttt{num}[4]$) will step to ($k \lhd \texttt{true}$).

$$
\begin{array}{lll}
(\texttt{num}\text{-}S) & k \rhd \texttt{num}[n] & \mapsto \quad k \lhd \texttt{num}[n] \\
(\texttt{cmp}\text{-}S_1) & k \rhd \texttt{cmp}[\mathit{cmp}](e_1; e_2) & \mapsto \quad k; \texttt{cmp}[\mathit{cmp}](\square; e_2) \rhd e_1 \\
(\texttt{cmp}\text{-}S_2) & k; \texttt{cmp}[\mathit{cmp}](\square; e_2) \lhd v_1 & \mapsto \quad k; \texttt{cmp}[\mathit{cmp}](v_1; \square) \rhd e_2 \\
(\texttt{cmp}\text{-}S_3) & k; \texttt{cmp}[\mathit{cmp}](\texttt{num}[n_1]; \square) \lhd \texttt{num}[n_2] & \mapsto \quad k \lhd n_1\ \mathit{cmp}\ n_2 \\
(\texttt{calc}\text{-}S_1) & k \rhd \texttt{calc}[\mathit{op}](e_1; e_2) & \mapsto \quad k; \texttt{calc}[\mathit{op}](\square; e_2) \rhd e_1 \\
(\texttt{calc}\text{-}S_2) & k; \texttt{calc}[\mathit{op}](\square; e_2) \lhd v_1 & \mapsto \quad k; \texttt{calc}[\mathit{op}](v_1; \square) \rhd e_2 \\
(\texttt{calc}\text{-}S_3) & k; \texttt{calc}[\mathit{op}](\texttt{num}[n_1]; \square) \lhd \texttt{num}[n_2] & \mapsto \quad k \lhd \texttt{num}[n_1\ \mathit{op}\ n_2] \quad (\text{if } n_1\ \mathit{op}\ n_2 \text{ is defined}) \\
(\texttt{calc}\text{-}F) & k; \texttt{calc}[\mathit{op}](\texttt{num}[n_1]; \square) \lhd \texttt{num}[n_2] & \mapsto \quad k \blacktriangleleft \quad (\text{if } n_1\ \mathit{op}\ n_2 \text{ is not defined}) \\
(\texttt{true}\text{-}S) & k \rhd \texttt{true} & \mapsto \quad k \lhd \texttt{true} \\
(\texttt{false}\text{-}S) & k \rhd \texttt{false} & \mapsto \quad k \lhd \texttt{false} \\
(\texttt{if}\text{-}S_1) & k \rhd \texttt{if}(e; e_t; e_f) & \mapsto \quad k; \texttt{if}(\square; e_t; e_f) \rhd e \\
(\texttt{if}\text{-}S_2) & k; \texttt{if}(\square; e_t; e_f) \lhd \texttt{true} & \mapsto \quad k \rhd e_t \\
(\texttt{if}\text{-}S_3) & k; \texttt{if}(\square; e_t; e_f) \lhd \texttt{false} & \mapsto \quad k \rhd e_f
\end{array}
$$

Sequences We are only using sequences to describe the *sequential* evaluation of our language; this means that if we look exclusively at this section then the semantics of sequences seem to be missing a critical rule for tabulation $\texttt{tab}[\tau](x.e; e_n)$.

$$
\begin{array}{lll}
(\texttt{seq}\text{-}S) & k \rhd \texttt{seq}(v_0, \ldots, v_{n-1}) & \mapsto \quad k \lhd \texttt{seq}(v_0, \ldots, v_{n-1}) \\
(\texttt{tab}\text{-}S) & k \rhd \texttt{tab}[\tau](x.e; e_n) & \mapsto \quad k; \texttt{tab}[\tau](x.e; \square) \rhd e_n \\
(\texttt{tab}\text{-}F) & k; \texttt{tab}[\tau](x.e; \square) \lhd \texttt{num}[n] & \mapsto \quad k \blacktriangleleft \quad (\text{if } n \leq 0) \\
(\texttt{sub}\text{-}S_1) & k \rhd \texttt{sub}(e; e_n) & \mapsto \quad k; \texttt{sub}(\square; e_n) \rhd e \\
(\texttt{sub}\text{-}S_2) & k; \texttt{sub}(\square; e_n) \lhd v & \mapsto \quad k; \texttt{sub}(v; \square) \rhd e_n \\
(\texttt{sub}\text{-}S_3) & k; \texttt{sub}(\texttt{seq}(v_0, \ldots, v_{n-1}); \square) \lhd \texttt{num}[i] & \mapsto \quad k \lhd v_i \quad (\text{if } 0 \leq i < n)
\end{array}
$$

$$
\begin{array}{lll}
(\texttt{sub}\text{-}F) & k; \texttt{sub}(\texttt{seq}(v_0, \ldots, v_{n-1}); \square) \lhd \texttt{num}[i] \quad \mapsto \quad k \blacktriangleleft & (\text{if } 0 > i \text{ or } i \geq n) \\
(\texttt{len}\text{-}S_1) & k \rhd \texttt{len}(e) \quad \mapsto \quad k; \texttt{len}(\square) \rhd e & \\
(\texttt{len}\text{-}S_2) & k; \texttt{len}(\square) \lhd \texttt{seq}(e_0, \ldots, e_{n-1}) \quad \mapsto \quad k \lhd \texttt{num}[n] & \\
(\texttt{showt}\text{-}S_1) & k \rhd \texttt{showt}(e; x.e_1; y_1.y_2.e_2) \quad \mapsto \quad k; \texttt{showt}(\square; x.e_1; y_1.y_2.e_2) \rhd e & \\
(\texttt{showt}\text{-}S_2) & k; \texttt{showt}(\square; x.e_1; y_1.y_2.e_2) \lhd \texttt{seq}(v) \quad \mapsto \quad k \rhd [v/x]e_1 & \\
(\texttt{showt}\text{-}S_3) & k; \texttt{showt}(\square; x.e_1; y_1.y_2.e_2) \lhd \texttt{seq}(v_0, \ldots, v_{n-1}) & (\text{if } n > 1 \text{ and } k = \lfloor n/2 \rfloor) \\
& \quad \mapsto \quad k \rhd [\texttt{seq}(v_0, \ldots, v_{k-1})/y_1][\texttt{seq}(v_k, \ldots, v_{n-1})/y_2]e_2 &
\end{array}
$$

2.2 Statics

There are three judgments giving the statics of states s. The judgment $s : \tau$ expresses that, if the state s ever returns a value to the empty stack $s = (\ \epsilon \lhd v\ )$, then v will be a value of type τ. In support of this judgment, we need two more judgments $k : \tau \Rightarrow \tau'$ and $f : \tau \Rightarrow \tau'$ which express that the stack k and the frame f transform values of type τ to values of type τ'.

$$\frac{k : \tau' \Rightarrow \tau \qquad \emptyset \vdash e : \tau'}{(\ k \rhd e\ ) : \tau}(S_1) \qquad \frac{k : \tau' \Rightarrow \tau \qquad \emptyset \vdash v : \tau' \qquad v\ \mathsf{val}}{(\ k \lhd v\ ) : \tau}(S_2) \qquad \frac{k : \tau' \Rightarrow \tau}{(\ k \blacktriangleleft\ ) : \tau}(S_3)$$

$$\frac{}{\epsilon : \tau \Rightarrow \tau}(K_1) \qquad \frac{k : \tau' \Rightarrow \tau_{\mathit{fin}} \qquad f : \tau \Rightarrow \tau'}{k; f : \tau \Rightarrow \tau_{\mathit{fin}}}(K_2)$$

The value judgment shows up in the typing rules for frames as it did in the statics rule for $\texttt{seq}(e_0, \ldots, e_{n-1})$:

$$\frac{x : \tau \vdash e_1 : \tau'}{\texttt{let}(\square; x.e_1) : \tau \Rightarrow \tau'}(\texttt{let}\text{-}F)$$

$$\frac{\emptyset \vdash e_2 : \tau'}{\texttt{ap}(\square; e_2) : \tau' \rightharpoonup \tau \Rightarrow \tau}(\texttt{apA}\text{-}F) \qquad \frac{\emptyset \vdash v_1 : \tau \rightharpoonup \tau' \qquad v_1\ \mathsf{val}}{\texttt{ap}(v_1; \square) : \tau \Rightarrow \tau'}(\texttt{apB}\text{-}F)$$

$$\frac{}{\texttt{pr[l]}(\square) : \tau_1 \times \tau_2 \Rightarrow \tau_1}(\texttt{prl}\text{-}F) \qquad \frac{}{\texttt{pr[r]}(\square) : \tau_1 \times \tau_2 \Rightarrow \tau_2}(\texttt{prr}\text{-}F)$$

$$\frac{\emptyset \vdash e_2 : \texttt{int}}{\texttt{cmp}[\mathit{cmp}](\square; e_2) : \texttt{int} \Rightarrow \texttt{bool}}(\texttt{cmpA}\text{-}F) \qquad \frac{v_1\ \mathsf{val} \qquad \emptyset \vdash v_1 : \texttt{int}}{\texttt{cmp}[\mathit{cmp}](v_1; \square) : \texttt{int} \Rightarrow \texttt{bool}}(\texttt{cmpB}\text{-}F)$$

$$\frac{\emptyset \vdash e_2 : \texttt{int}}{\texttt{calc}[\mathit{op}](\square; e_2) : \texttt{int} \Rightarrow \texttt{int}}(\texttt{calcA}\text{-}F) \qquad \frac{v_1\ \mathsf{val} \qquad \emptyset \vdash v_1 : \texttt{int}}{\texttt{calc}[\mathit{op}](v_1; \square) : \texttt{int} \Rightarrow \texttt{int}}(\texttt{calcB}\text{-}F)$$

$$\frac{\emptyset \vdash e_t : \tau \qquad \emptyset \vdash e_f : \tau}{\texttt{if}(\square; e_t; e_f) : \texttt{bool} \Rightarrow \tau}(\texttt{if}\text{-}F)$$

Task 2.1 (5%). The statics for frames in the sequence-manipulating fragment of the language – that is, the frames $\texttt{tab}[\tau](x.e; \square)$, $\texttt{sub}(\square; e_n)$, $\texttt{sub}(v; \square)$, $\texttt{len}(\square)$, and $\texttt{showt}(\square; x.e_1; y_1.y_2.e_2)$ – have been omitted. Give these rules such that progress and preservation (discussed in the next section) will hold. □

Task 2.2 (5%). We will not require you to implement the statics of frames and control machine states in this homework. Implementing these statics, however, gives you a powerful invariant for checking the well-formedness of intermediate states. This might help you find errors in the implementation of the dynamics (which will show up as failures of preservation).

In this task, we will ask you to think through one aspect of this optional implementation. If you do set `TypeChecker.canCheckStates` to `true` in `typechecker.sml`, then the top-level interpreter's `step` command will perform typechecking on intermediate states. In order to correctly implement `TypeChecker.checkstate: State.t -> Type.t`, you will need a helper function that handles the statics of frames. **[Bug fixing update: this function only needs to correctly handle states** ($k \rhd e$) **and** ($k \lhd v$)**, not failure states** ($k \blacktriangleleft$)**.]** For each of the following options, explain whether or not it would be implementable and usable as a helper function to the `TypeChecker.checkstate` function.

- `frametype: Frame.t -> Tp.t * Tp.t -> unit`
  where ($\texttt{frametype}\ f\ (\tau, \tau')$) will run without error iff $f : \tau \Rightarrow \tau'$
- `frametype: Frame.t -> Tp.t -> Tp.t`
  where ($\texttt{frametype}\ f\ \tau) = \tau'$ iff $f : \tau \Rightarrow \tau'$
- `frametype: Frame.t -> Tp.t -> Tp.t`
  where ($\texttt{frametype}\ f\ \tau') = \tau$ iff $f : \tau \Rightarrow \tau'$
- `frametype: Frame.t -> Tp.t * Tp.t`
  where ($\texttt{frametype}\ f) = (\tau, \tau')$ iff $f : \tau \Rightarrow \tau'$

If you indicate that a particular version of `frametype` would work, then give pseudo-code or code for indicating how it would be used to check the type of states – describing the ($k \rhd e$) case will be sufficient.

2.3 Safety

We'd like to say that the standard progress and preservation formulation of our language holds:

Theorem 1 (Preservation)**.** *If $s : \tau$ and $s \mapsto s'$ then $s' : \tau$.*

Theorem 2 (Progress)**.** *If $s : \tau$ then either there exists an s' such that $s \mapsto s'$ or s* final.

However, progress only holds true if we carefully formulate the s final judgment.

Task 2.3 (5%). Give a definition of the s final judgment such that the progress theorem holds. Make sure to account for all the well-formed states that cannot step according to the dynamics we've presented so far. Your final judgment shouldn't declare any states to be final if they can take a step, so you can't just write

$$\frac{}{s\ \text{final}}(\textit{this is bogus})$$

even though this would certainly ensure that the progress theorem holds.

Task 2.4 (10%). Consider each of the following changes being made to our language definition *independently and one at a time, **not** cumulatively*.

1. Adding a e_1 val premise to `pair-val`.

2. Removing the e_i val premises from seq-I_1.
3. Making the right-hand side of prl-S_2 ($k \lhd e_1$) instead of ($k \rhd e_1$).
4. Making the right-hand side of prl-S_2 ($k \rhd e_2$) instead of ($k \rhd e_1$).
5. Removing the premise from S_3.
6. Removing the v_1 val premise from apB-F.
7. Adding a e_2 val premise to cmpA-F.

For each potential change, do one of the following:

- State that it does not break preservation and progress for the language. If you feel it breaks the language in some way that isn't captured by progress and preservation, you can state this, but it isn't necessary to do so.
- State that it breaks preservation and give an s such that $s : \tau$ and $s \mapsto s'$ but it is not the case $s' : \tau$. (If it will not be obvious to us why $s : \tau$ or why $s \mapsto s'$, make this clear.)
- State that it breaks progress and give an s such that $s : \tau$ but s neither steps nor takes a value. (If it will not be obvious to us why $s : \tau$, make this clear.)

2.4 Implementation

The top-level for sequential evaluation can be invoked by running `Seq.repl()`.

Task 2.5 (15%). Implement the sequential dynamics of control stacks in `dynamics.sml` as the function `trystep`. Your code should be clear and well-organized in addition to being correct. Running `trystep` s should return:

- `Steps` s' if $s \mapsto s'$,
- `Value` v if $s = (\ \epsilon \lhd v\ )$,
- `Tabulate` $(n, k, (x, e))$ if $s = (\ k; \texttt{tab}[\tau](x.e; \Box) \lhd \texttt{num}[n]\ )$, and
- `Err` if $s = (\ k \blacktriangleleft\ )$. □

Here is an example of running the sequential abstract machine toplevel interpreter:

```
$ rlwrap sml -m sources.cm
- Seq.repl ();
->step 1+2 == 5;
Statics: exp has type bool
 --> emp; Cmp[Eq](-,Num[5])
       |> Calc[Plus](Num[1], Num[2])
->step;
 --> emp; Cmp[Eq](-,Num[5]); Calc[Plus](-,Num[2])
       |> Num[1]
->step;
 --> emp; Cmp[Eq](-,Num[5]); Calc[Plus](-,Num[2])
```

```
       <| Num[1]
->eval;
 False VAL
->eval let val S = seq(8,6,4,2,0) in S[0] + S[3] end;
Statics: exp has type int
 Num[10] VAL
```

3 Parallel dynamics

We describe the global state of a parallel computation as a mapping from symbols a_i to abstract machines s_i. These abstract machines collectively capture the steps that need to happen to finish the global computation. These global states have the form $\nu\Sigma\{\mu\}$, where the signature Σ associates types to each symbol:

$$\Sigma = a_1 \sim \tau_1, \ldots, a_n \sim \tau_n$$

and μ associates an abstract machine with each symbol:

$$\mu = a_1 \hookrightarrow s_1 \otimes \ldots \otimes a_n \hookrightarrow s_n$$

We treat the symbols a_i as being *bound* in Σ and therefore subject to renaming, though this renaming will not be visible in our implementation – as with variables (or assignables in Algol) we will make sure each newly generated symbol is given a fresh name when we call the `Sym.newsym` function.

A local transition deals only with a restricted portion of the memory. The easy case of a local transition is a state that is able to take a step according to sequential semantics of control stacks:

$$\frac{s \mapsto s'}{\nu a \sim \tau\{a \hookrightarrow s\} \mapsto_{loc} \nu a \sim \tau\{a \hookrightarrow s'\}}(\texttt{locstep-step})$$

A different local transition finally gives us a way to consider the previously-unhandled abstract machine state $(\ k; \texttt{tab}[\tau](x.e; \square) \lhd \texttt{num}[n]\ )$, which needs to spawn n computations that will be evaluated in parallel.

$$\frac{n > 0}{\begin{array}{c}\nu a \sim \tau'\{a \hookrightarrow (\ k; \texttt{tab}[\tau](x.e; \square) \lhd \texttt{num}[n]\ )\} \mapsto_{loc} \\ \nu a \sim \tau', a_0 \sim \tau, \ldots, a_{n-1} \sim \tau\{a \hookrightarrow (\ k\ \mathsf{awaiting}[a_0, \ldots, a_{n-1}]\ ) \otimes \mu\}\end{array}}(\texttt{locstep-fork})$$

where $\mu = a_0 \hookrightarrow (\ \epsilon \rhd [\texttt{num}[0]/x]e\ ) \otimes \ldots \otimes a_{n-1} \hookrightarrow (\ \epsilon \rhd [\texttt{num}[n-1]/x]e\ )$. This rule also introduces a new form of state $s = (\ k\ \mathsf{awaiting}[a_0, \ldots, a_{n-1}]\ )$, which represents a stack that expects to receive a length-n sequence.

A sequence can be created and returned to the stack when the computations associated with symbols $a_0, \ldots, a_{n-1}$ eventually produce values, which happens with our final local transition rule:

$$\frac{}{\begin{array}{c}\nu a \sim \tau', a_0 \sim \tau, \ldots, a_{n-1} \sim \tau\{a \hookrightarrow (\ k\ \mathsf{awaiting}[a_0, \ldots, a_{n-1}]\ ) \otimes \mu\} \\ \mapsto_{loc} \nu a \sim \tau'\{a \hookrightarrow (\ k \lhd \texttt{seq}(v_0, \ldots, v_{n-1})\ )\}\end{array}}(\texttt{locstep-join})$$

where $\mu = a_0 \hookrightarrow (\ \epsilon \lhd v_0\ ) \otimes \ldots \otimes a_{n-1} \hookrightarrow (\ \epsilon \lhd v_{n-1}\ )$.

Global transitions are made by partitioning the global state into pieces and performing some number of local transitions on those pieces:

$$\frac{\nu\Sigma_A\{\mu_A\} \mapsto_{loc} \nu\Sigma'_A\{\mu'_A\} \qquad \ldots \qquad \nu\Sigma_Z\{\mu_Z\} \mapsto_{loc} \nu\Sigma'_Z\{\mu'_Z\}}{\nu\Sigma_A, \ldots, \Sigma_Z, \Sigma\{\mu_A \otimes \ldots \otimes \mu_Z \otimes \mu\} \mapsto_{glo} \nu\Sigma'_A, \ldots, \Sigma'_Z, \Sigma\{\mu'_A \otimes \ldots \otimes \mu'_Z \otimes \mu\}}$$

This global transition rule is highly nondeterministic in general.

Task 3.1 (10%). Implement the parallel dynamics in `dynamics.sml`.

- The function `ready` should calculate all the symbols that are ready to take a local transition.

  Given $\mu = (a_1 \hookrightarrow s_1) \otimes \ldots \otimes (a_n \hookrightarrow s_n)$, we can say that a_i is *ready* if

  - $s_i = (\ k \rhd e\ )$,
  - $s_i = (\ k \lhd v\ )$ and $k \neq \epsilon$, or if
  - $s_i = (\ k\ \mathsf{awaiting}[b_0, \ldots, b_{n-1}]\ )$ and for each of the b_i, there is a mapping $b_i \hookrightarrow (\ \epsilon \lhd v_i\ ) \in \mu$

  Every possible global transition step can be uniquely specified as some subset of the ready symbols.

- The function `trystep_par` should take a well-formed execution context and a list of ready symbols and should execute a step of the global transition.

Here is an example of running the parallel abstract machine toplevel interpreter:

```
$ rlwrap sml -m sources.cm
- Par.repl ();
->use "parfact.par";
Statics: exp has type int
 --> {
        (id: 0)
        init@1 ~~>
          emp
            |> Let(Fix[parr(int, int)](fact@11. Lam[int](x@12. If(Cmp[Eq](x@...
     }
    *** Local steps taken (work): 0
    *** Global steps taken: 0
->eval;
 Num[3623758] VAL
    *** Local steps taken (work): 1329
    *** Global steps taken: 254
->eval let val S = [ x * x * x | x < 300] in S[12] end;
Statics: exp has type int
 Num[1728] VAL
    *** Local steps taken (work): 2711
    *** Global steps taken: 20
->load [ x | x < 5 ];
Statics: exp has type seq(int)
 --> {
        (id: 0)
        init@31 ~~>
          emp
            |> Tab(x@386. x@386, Num[5])
     }
    *** Local steps taken (work): 0
    *** Global steps taken: 0
->step; step; step;
(...there's more in par-interpreter-session.txt...)
```

4 Parallel Programming

Different implementations of sequences can have different costs in terms of their work (the amount of time computation takes with one processor) and span (the amount of time computation takes with an unlimited number of processors). For the code in this section, which you will write in `mergesort.par`, we will assume the 15-210 ArraySequence implementation of sequences, meaning that getting a single element of an sequence with $e[e_n]$, finding out the length of an sequence with $\texttt{len}(e)$, and case analyzing on whether a sequence has length-1 or not with $\texttt{showt}\ e\ \{\texttt{elt}\ x \Rightarrow e_1 \mid \texttt{node}\ y_1\ y_2 \Rightarrow e_2\}$ are all constant-work and constant-depth operations. Sequence tabulation can be done entirely in parallel, so its work just the sum of the work needed to calculate each sequence element, and its span is the maximum of the spans needed to calculate each sequence element.

Task 4.1 (5%). The `map` and `cat` operations described in *PFPL* can be implemented with appropriate work (linear) and span (constant) using the tabulation primitive. Implement `map` and `cat` for $\texttt{seq}(\texttt{int})$.

Task 4.2 (5%). Write a parallel reduce on `int` sequences. Consider a sequence of integers and an associative binary operation called $\circledcirc$. Then, the expected behavior of reduce is as follows:

$$\texttt{reduce}\ (\circledcirc)\ (\texttt{seq}(v_0, \ldots, v_{n-1})) = \texttt{seq}(v_0) \circledcirc \texttt{seq}(v_1) \circledcirc \ldots \circledcirc \texttt{seq}(v_{n-2}) \circledcirc \texttt{seq}(v_{n-1})$$

Since $\circledcirc$ is associative, we can parenthesize any way we want. In particular, we can parenthesize this as a balanced binary tree:

$$\Big(\Big(\big(\texttt{seq}(v_0) \circledcirc \texttt{seq}(v_1)\big) \circledcirc \big(\texttt{seq}(v_2) \circledcirc \texttt{seq}(v_3)\big)\Big) \circledcirc \Big(\ldots\Big)\Big) \circledcirc \Big(\ldots\Big)$$

Then for each level of this tree, we can compute all the nodes at that level in parallel.

This is almost trivial to implement with $\texttt{showt}\ e\ \{\texttt{elt}\ x \Rightarrow e_1 \mid \texttt{node}\ y_1\ y_2 \Rightarrow e_2\}$, though you have to take care to make sure evaluation actually happens in parallel. For full credit, implement `reduce` without `showt` (a correct `showt` solution will get 3 out of 5 points).

Task 4.3 (5%). Assume the length of the sequence is a power of 2. What is the work and span of `reduce` if the associative binary operation has constant work and span? What if it has constant span but linear work (like `cat`)? What if it has $O(\log n)$ span and linear work (like `merge`)?

Task 4.4 (5%). Implement a semantically correct merge operation on sorted sequences. It is not known how to implement `merge` with $O(n)$ work and $O(\log n)$ span using the primitives and cost semantics we have established so far. We'll be looking for the best answers and may award extra credit (5-10 points) to particularly efficient solutions, but your solution doesn't have to be work efficient (though the work should be in $O(n^2)$) or parallelizable. It's fine to use Google and outside sources (15-150 notes, Guy Blelloch's dissertation...) for reference and inspiration, but the usual policy on peer-to-peer collaboration applies.

Task 4.5 (5%). Implement parallel mergesort using the other functions you have written (use `showt`). Make sure the two halves of a sequence are actually sorted in parallel according to the semantics.

5 Scheduling

Take a look at the file `tex/par-interpreter-session.txt` in the handout code. Running `step` or `eval` without any arguments always causes the maximum number of local transitions to accompany every global transition, simulating execution with an unlimited number of processors. But there is also an option for selecting exactly which local transitions should take a step.

```
->step <0,3,4>; (* Picks the local transitions identified by 0, 3, and 4 *)
```

By manually performing these selections such that at most p local transitions happen at any time, you would be, in effect, acting like a *scheduler* that is coordinating the available work among p different processors.

The overall progression of a parallel computation can be viewed as a *series-parallel* diagram like the one to the right, where the dots represent a *local* transition that must be made for the computation to complete. Any global state that could arise during evaluation corresponds to a cut through this graph. The cut labeled **1** to the right corresponds to a global state where there are four mappings from assignables to states. Three of these mappings (call them $a_1 \hookrightarrow s_1$, $a_2 \hookrightarrow s_2$, and $a_3 \hookrightarrow s_3$) would correspond to the states that are ready to make local transitions labeled o, g, and h, respectively. The transitions o and g are sequential transitions by (`locstep-step`), and the transition h is a forking transition (`locstep-fork`). The fourth mapping would be of the form $a_0 \hookrightarrow (\ k$ awaiting$[a_1, a_2, a_3]\ )$, as it is a state awaiting the termination of these three other computations; the next local step that this state will take corresponds to the node labeled q.

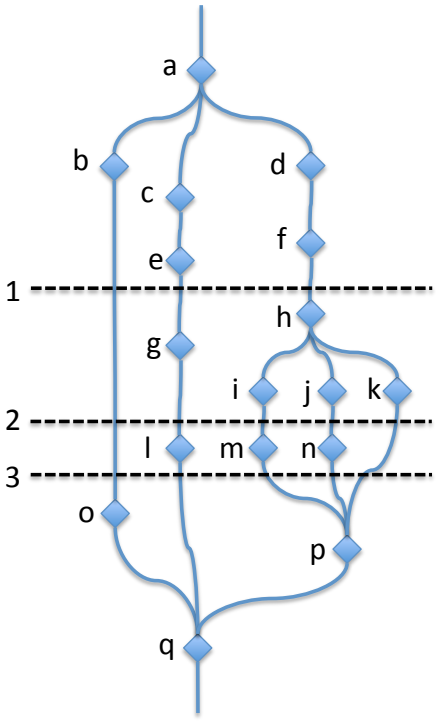


Figure 2: Series-parallel diagram

5.1 Scheduling strategies

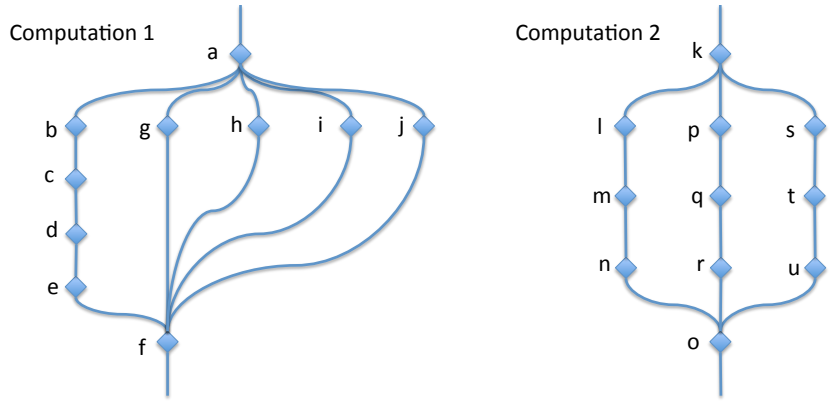
A scheduler provides some way of mapping the work that needs to be done onto a fixed number of processors that are available to do work. *Greedy* schedulers are ones that always keep processors working if there's available work to do. The most obvious greedy scheduling strategies that have been studied in practice are p-BFS and p-DFS (breadth-first and depth-first scheduling on p processors). Breadth-first scheduling is a simple strategy that can be implemented with a queue: if there's more available work than available parallelism, always prioritize one of the units of work that has been waiting longest to be scheduled. Depth-first scheduling always schedules the *leftmost* work from the perspective of the series-parallel diagram; visually this is rather clear, but the data structure to implement it is a bit trickier.

For the computation associated with the series-parallel diagram in Figure 2, Figure 3 gives an example of how scheduling would proceed under strict 2-DFS and 2-BFS scheduling.

	2-DFS Scheduling		2-BFS Scheduling	
Round #	Scheduled	Also ready	Scheduled	Also ready
1	a		a	
2	b, c	d	b, c	d
3	o, e	d	d, o	e
4	g, d		e, f	
5	l, f		g, h	
6	h		l, i	j, k
7	i, j	k	j, k	m
8	m, n	k	m, n	
9	k		p	
10	p		q	
11	q			

Figure 3: Scheduling the diagram in Figure 2

Task 5.1 (5%). For both of these computations, give the order that work would be scheduled on 2 processors under breadth-first and depth-first scheduling.



This task shows that, BFS and DFS can have rather different performance depending on the shape of the computation; however, any greedy scheduler will take, at worst, twice the number of transitions that the best greedy scheduler would take. In practice, BFS and DFS scheduling (and more advanced variants, like *work stealing* schedulers, that we won't consider here) are important because they make a huge difference in space usage; unfortunately, this is outside the scope of this assignment. □

5.2 Implementation

In `scheduler.sml`, you will implement a centralized scheduler for p processors with processor IDs $0, \ldots, p-1$. Your implementation doesn't have to be any particular strategy (p-BFS, p-DFS, etc), but it should be greedy: if there's work to do and a processor is known to be idle, work should be scheduled on that processor.

Unlike our more theoretical treatments of scheduling, we won't necessarily try to reschedule everything at every single time-step. Consider our original series-parallel diagram example, for instance. If we'd

already assigned a processor to do work c, then we should probably just let it continue to run the sequential computation, doing work e, g, and l. The scheduler will only be called when a processor runs out of work to do, generating one of three messages:

- `Finished` v – the processor has finished sequentially evaluating the work it was given, and the result was a value.
- `Fork` $(n, k, (x, e))$ – the processor reached the state (k; `tab`$[\tau](x.e; \Box) \lhd$ `num`$[n]$) and now needs n sub-computations to be run.
- `NotFinished` s – the processor didn't finish sequentially evaluating the work it was given, but it was told to yield to the scheduler after taking some fixed number of steps. (You can also tell processors to run as long as necessary, in which case you'll never get this message.)

Whenever the scheduler is called, it can either declare the global computation finished and return a value, or it can return a list of processor IDs and states $[(p_0, s_0), \ldots, (p_k, s_k)]$ that should begin computing. The processors p_i in this returned list must currently be idle. In order to do this correctly, you'll need to hang on to three key pieces of state:

- Information about which processors are currently known to be idle. The starter code suggests storing this in an array `currently_working`. If `NONE` $=$ `Array.sub(!currently_working,` k`)`, it means that the processor k is known to be idle.
- Information about which ready pieces of work are waiting to be scheduled (because at some point there enough processors). The starter code suggests storing this in a list `unscheduled_work`. If you follow this, it means that what you implement won't quite be p-DFS. You're welcome to modify this to implement something closer to p-BFS or true p-DFS.
- Information about *why* you're running, or waiting to run, a given computation. Every computation except for the initial one is trying to compute the i^{th} element of some sequence. Taking this into account, instead of associating every computation with a symbol, we will associate every computation with a *destination*, either `Initial` for the first computation or `WriteTo`$\{loc, index = i\}$ for every other computation.

Whenever a fork spawns n new computations, the computation ($stack$ awaiting$[a_0, \ldots, a_{n-1}]$) is represented by an object loc = `Mem`$\{dest, waiting\_on, stack, seq\}$, where seq is initially a length-n array containing all `NONE` values and $waiting\_on$ is a reference initially containing n. Whenever a sub-computation returns a value v, you should write `SOME` v to the appropriate index of the array seq and decrement $waiting\_on$. If $waiting\_on$ has been decremented all the way to 0, then the array seq contains n values `SOME` $v_0, \ldots,$ `SOME` v_{n-1}, and it is possible to immediately re-schedule the computation ($stack \lhd$ `seq`$(v_0, \ldots, v_{n-1})$) that has been waiting by adding it back to the `unscheduled_work` worklist. This newly-rescheduled computation's destination, $dest$, was stored as a part of loc.

Task 5.2 (20%). Implement the scheduler in `scheduler.sml`. Upon receiving a message from the processor k, the scheduler should:

1. Grab the destination associated with k's computation from `currently_working`$[k]$, and then mark processor k as idle by setting `currently_working`$[k]$ to `NONE`.

2. If k returned a value (`Finished`), store the resulting value in the right place – or terminate the computation if it's all finished. If storing the resulting value re-enables a suspended computation, add that suspended computation to the worklist.

3. If the processor returned with more work to do (`NotFinished` or `Fork`), add all the unfinished work to the worklist.

4. Reschedule: remove jobs from the worklist and assign them to idle processors until either you run out of available processors or until the worklist is empty.

You're welcome to modify the given data structures at your own risk as long as you do not change the interface or reference the sequential dynamic semantics. □

Rather than giving a REPL, the structure `Simulate` in `simulate.sml` provides facilities for running and testing the scheduler on the command line. The `step_limit` argument determines how long a processor will run a sequential computation before yielding to the scheduler with a `NotFinished` message, and `step_limit` = `NONE` means it will only yield when it is finished.

```
$ rlwrap sml -m sources.cm
- Simulate.evalFile { num_procs = 5, step_limit = NONE, filename = "parfact.par" };
one new task scheduled
<OOOOO> => <XOOOO>
Processor 0 invoking scheduler (fork 11)... 5 new tasks scheduled
<!OOOO> => <XXXXX>
Processor 4 invoking scheduler (done)... one new task scheduled
<XXXX!> => <XXXXX>
Processor 3 invoking scheduler (done)... one new task scheduled
<XXX!X> => <XXXXX>
Processor 2 invoking scheduler (done)... one new task scheduled
<XX!XX> => <XXXXX>
Processor 1 invoking scheduler (done)... one new task scheduled
<X!XXX> => <XXXXX>
Processor 0 invoking scheduler (done)... one new task scheduled
<!XXXX> => <XXXXX>
Processor 0 invoking scheduler (done)... one new task scheduled
<!OOOO> => <XOOOO>
Processor 1 invoking scheduler (done)... no new tasks scheduled
<X!OOO> => <XOOOO>
Processor 2 invoking scheduler (done)... no new tasks scheduled
<XO!OO> => <XOOOO>
Processor 3 invoking scheduler (done)... no new tasks scheduled
<XOO!O> => <XOOOO>
Processor 4 invoking scheduler (done)... no new tasks scheduled
<XOOO!> => <XOOOO>
Processor 0 invoking scheduler (done)... one new task scheduled
<!OOOO> => <XOOOO>
Processor 0 invoking scheduler (done)... all done.
val it = {res=-,steps=304,work=1329} : {res:?.t, steps:int, work:int}
- print (Term.toString (#res it)^"\n");
Num[3623758]
```

By default, the simulator will print a message before and after every invocation of the scheduler, reporting on which processors are busy (`X`) and which are idle (`O`). To turn this off, you can set the flag `Simulate.verbose := false`.

The work (total number of local transitions) reported by this simulator should be the same as the work reported by the parallel semantics, but even if you make more than enough processors available, the number of global transitions may be slightly different. This is both because the scheduler now effectively performs the (`locstep-join`) local transitions as soon as they are possible and because finished computations need to wait for an additional global transition before they report back to the scheduler that they've finished.

A Putting The Code Together

As usual, you can compile your files for using `CM.make "sources.cm"`, and examples of using the top-level interpreter to run sequential and parallel dynamics have been given above.

A.1 Interpreter

The syntax for each term construct is as close as possible to the concrete syntax mentioned for it. (The concrete syntax is the second column in the table which introduces the syntax for the language.) We provide below the grammar that the interpreter accepts.

```
ident ::= (* a letter followed by alphanumeric characters, _, or ' *)
numeral ::= (* a series of digits *)
strconst ::= (* a standard C-escaped string like "foo" or "why \"hello\"" *)

seqcommand ::=
  eval <exp>;
| eval;
| step <exp>;
| step;
| load <exp>;
| use <stringconst>;

parcommand ::=
  eval <exp>;
| eval;
| step < <exps> >; (* <exps> should be comma-separated numbers *)
| step;            (* all possible local transitions happen *)
| load <exp>;
| use <stringconst>;

ty ::= int | bool | <ty> -> <ty> | <ty> * <ty> | seq <ty>

exp ::=
  let <decls> in <exp> end
| fn (<ident> : <ty>) <exp>
| <exp>(<exp>)
| fix <ident> : <ty> is <exp>
| <<exp>, <exps>>
| <exp>.l
```

```
| <exp>.r
| <numeral>
| <exp> == <exp>
| <exp> != <exp>
| <exp> >= <exp>
| <exp> <= <exp>
| <exp> + <exp>
| <exp> - <exp>
| <exp> * <exp>
| <exp> / <exp>
| <exp> % <exp>
| true
| false
| if <exp> then <exp> else <exp>
| seq(<exps>)
| [ <exp> | <ident> < <exp> ]
| <exp> [ <exp> ]
| len <exp>
| showt <exp> { elt <ident> => <exp> | node <ident> <ident> => <exp> }

exps ::= <exp> | <exp>, <exps>

decls ::= <decl> | <decl> <decls>

decl ::= val <ident> = <exp>
```

A.2 Reference implementation

You can use the reference interpreter to test your code and help debug your interpreter.

The reference interpreter can be loaded like this:

```
$ rlwrap sml @SMLload=/afs/andrew/course/15/312/bin/ref_impl_par
- Seq.repl ();
- Par.repl ();
```