A Cost Semantics for Parallelism

September 14, 2017

Functional Core Language

We will work with a minimal functional language for expressing parallel algorithms that act on values of a fixed collection of recursive data types.

Syntax of types:

τ	$::=$	t_i	ith data type
		$\tau_1 \times \cdots \times \tau_n$	n-tuple type
		$\tau_1 \rightharpoonup \tau_2$	partial functions

Syntax of expressions and patterns:

e	$::=$	x	variable
		$c\,e$	constructor instance
		$\texttt{case}\, e \,\{\, c_1\, p_1 \hookrightarrow e_1 \mid \cdots \mid c_n\, p_n \hookrightarrow e_n \,\}$	case analysis
		$\langle e_1, \ldots, e_n \rangle$	n-tuple
		$\texttt{fun}\, x(p)\ \texttt{is}\ e$	recursive function
		$e_1(e_2)$	application
		$\texttt{error}$	error
p	$::=$	x	variable pattern
		$\_$	wildcard pattern
		$\langle p_1, \ldots, p_n \rangle$	tuple pattern

Substitution of a closed value, v, for free occurrences of x in e, written $[v/x]e$, is defined as usual. It amounts to replacing free occurrences of x

with v, there being no possibility of capture. Substitution is extended to patterns, written $[\![v/p]\!]e$, as follows:

$$[\![v/x]\!]e = [v/x]e$$
$$[\![v/\_]\!]e = e$$
$$[\![\langle v_1, \ldots, v_n\rangle/\langle p_1, \ldots, p_n\rangle]\!]e = [\![v_1/p_1]\!] \ldots [\![v_n/p_n]\!]e$$

The idea is that tuples are broken apart during substitution, so that, for example, substitution of $\langle v_1, \ldots, v_n\rangle$ for the pattern $\langle x_1, \ldots, x_n\rangle$ in an expression e is tantamount to replacing each x_i with v_i within e.

We assume given a collection of m data types

$$\texttt{data}\, t_i\, \texttt{is}\, c_{i1}\, \texttt{of}\, \tau_{i1} \mid \cdots \mid c_{in_i}\, \texttt{of}\, \tau_{in_i} \quad (1 \leq i \leq m, m \geq 0).$$

Among these we assume the following data types as given:[1]

1. `data void is` *(no constructors)*
2. `data bool is true | false`
3. `data nat is zero | succ of nat`
4. `data list is nil | cons of bool × list`.

The *cost semantics* is given by the following judgments:

e val	e is a value (that is, it is fully evaluated)
$e \Downarrow v\ [w; s]$	e evaluates to v with work w and span s
$e \Uparrow\ [w; s]$	e aborts with an error with work w and span s
$e \Updownarrow\ [w; s]$	e either evaluates or aborts with work w and span s

These judgements are *inductively defined* by a collection of rules of the form

$$\frac{J_1 \quad \ldots \quad J_n}{J}$$

where $n \geq 0$, and J and each of the J_i's are judgments. Informally, the meaning of the rule is that J holds whenever $J_1, \ldots, J_n$ all hold. Importantly, a judgment J is said to hold exactly when it can be derived by composing rules that end with J.[2]

[1] An omitted "of" specification means the same as saying `of unit`.

[2] This is what makes it an inductive definition.

For the sake of readability, the rules are given in three parts, those that define values, those that define evaluation, and those that define error propagation.

The rules defining what are the values are as follows:

$$\frac{e\ \mathsf{val}}{c\,e\ \mathsf{val}} \qquad \frac{}{\mathtt{fun}\,x(p)\ \mathtt{is}\ e\ \mathsf{val}}$$

$$\frac{e_1\ \mathsf{val} \quad \ldots \quad e_n\ \mathsf{val}}{\langle e_1, \ldots, e_n\rangle\ \mathsf{val}}$$

We often use the letter v to stand for an expression that is also a value according to these rules.

The rules defining evaluation are as follows:

$$\frac{e\ \mathsf{val}}{e \Downarrow e\ [1;1]}$$

$$\frac{e \Downarrow v\ [w;s] \quad \neg(e\ \mathsf{val})}{c\,e \Downarrow c\,v\ [w+1;s+1]}$$

$$\frac{e \Downarrow c_j\,v_j\ [w;s] \quad [\![v_j/p_j]\!]e_j \Downarrow v'_j\ [w_j;s_j]}{\mathtt{case}\,e\,\{\,c_1\,p_1 \hookrightarrow e_1 \mid \cdots \mid c_n\,p_n \hookrightarrow e_n\,\} \Downarrow v'_j\ [w+w_j+1;\max(s,s_j)+1]}$$

$$\frac{e_1 \Downarrow v_1\ [w_1;s_1] \quad e_2 \Downarrow v_2\ [w_2;s_2] \quad [\![v_2/p]\!][v_1/x]e \Downarrow v\ [w;s] \quad v_1 = \mathtt{fun}\,x(p)\ \mathtt{is}\ e}{e_1(e_2) \Downarrow v\ [w_1+w_2+w+1;\max(s_1,s_2)+s+1]}$$

$$\frac{e_1 \Downarrow v_1\ [w_1;s_1] \quad \ldots \quad e_n \Downarrow v_n\ [w_n;s_n] \quad \neg(\langle e_1, \ldots, e_n\rangle\ \mathsf{val})}{\langle e_1, \ldots, e_n\rangle \Downarrow \langle v_1, \ldots, v_n\rangle\ [1+\textstyle\sum_{i=1}^{n} w_i;1+\max_{i=1}^{n} s_i]}$$

Thus, values evaluate to themselves with unit work and span.[3] Constructed expressions that are not already evaluated add unit work and span to account for creating the value. Case analysis imposes unit work and span to dispatch on the constructor of the value. Similarly, application charges one unit of work and span for the call to the function and its return. Creating a tuple imposes an additional unit cost for creating the tuple.[4]

[3] For this to be realistic imposes the requirement on an implementation that a constructed value, no matter how large, be recognizable in constant time. Standard implementation methods satisfy this requirement.

[4] It might be argued that the additional cost should be n, the number of components of the tuple, but since n is constant (not dependent on input), it is sufficient to charge one unit of cost, which is accurate up to a scale factor.

It might be thought that errors simply abort the computation, and that's all there is to be said. But, after all, an error is the outcome of a computation, and we must account for the effort of computing it. Interestingly, the cost of an error is markedly different in a parallel language than in a sequential one. Consider the evaluation of an n-tuple expression. In a sequential language the components are evaluated from left to right, and the computation of the tuple stops as soon as an error occurs. In particular those components not yet evaluated at the point of the error need never be evaluated, and hence impose no cost. In a parallel language, however, there is no well-defined notion of "stopping early" in this sense. All components are evaluated simultaneously, and this cost is incurred *even if* one of the components incurs an error. The error is propagated upwards, but only after all other components have finished evaluating (perhaps incurring an error).

The rules governing error propagation are as follows:

$$\frac{}{\texttt{error} \Uparrow [1;1]}$$

$$\frac{e \Uparrow [w;s]}{c\,e \Uparrow [w+1;s+1]}$$

$$\frac{e \Uparrow [w;s]}{\texttt{case}\, e \,\{\, c_1\, p_1 \hookrightarrow e_1 \mid \cdots \mid c_n\, p_n \hookrightarrow e_n \,\} \Uparrow [w+1;s+1]}$$

$$\frac{e \Downarrow c_j\, v_j\ [w;s] \quad [\![v_j/p_j]\!]e'_j \Uparrow [w_j;s_j]}{\texttt{case}\, e \,\{\, c_1\, p_1 \hookrightarrow e_1 \mid \cdots \mid c_n\, p_n \hookrightarrow e_n \,\} \Uparrow [w+w_j+1;s+s_j+1]}$$

$$\frac{e_1 \Uparrow [w_1;s_1] \quad e_2 \Updownarrow [w_2;s_2]}{e_1(e_2) \Uparrow [1+w_1+w_2+1;1+\max(s_1,s_2)+1]}$$

$$\frac{e_1 \Updownarrow [w_1;s_1] \quad e_2 \Uparrow [w_2;s_2]}{e_1(e_2) \Uparrow [w_1+w_2+1;\max(s_1,s_2)+1]}$$

$$\frac{e_1 \Updownarrow [w_1;s_1] \quad \ldots \quad e_j \Uparrow [w_j;s_j] \quad \ldots \quad e_n \Updownarrow [w_n;s_n]}{\langle e_1,\ldots,e_n\rangle \Uparrow [1+\sum_{i=1}^n w_i; 1+\max_{i=1}^n s_i]}$$

Some remarks:

1. Define $\lambda\, x.e$ to mean $\texttt{fun}\ \_(x)\ \texttt{is}\ e$, with an irrelevant name for the function itself, and define $\texttt{let}\ x\ \texttt{be}\ e_1\ \texttt{in}\ e_2$ to mean $(\lambda\, x.e_2)(e_1)$. As an exercise, derive the cost semantics for $\texttt{let}\ x\ \texttt{be}\ e_1\ \texttt{in}\ e_2$ from this definition.

2. Define $\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ to mean $\texttt{case}\ e\ \{\ \texttt{true}\ \_ \hookrightarrow e_1 \mid \texttt{false}\ \_ \hookrightarrow e_2\ \}$. Derive the cost semantics for the conditional from this definition.

3. It is possible to reduce everything to just λ-abstraction and application (and variables) using (a) Church's encodings of data, and (b) Kleene's **Y** combinator.

4. Consider the possibility of there being more than one form of error. How must the cost semantics change to account for this? *Hint*: think carefully about n-tuples!

The cost semantics defines an abstract notion of work and span in the sense that the units of work represent fixed-cost computations that have to be performed, but ignoring the specifics of exactly what is involved in each case. The true cost is a constant factor of the abstract cost, a detail that we shall ignore in this overview.

The abstract cost may be related to the true cost on a p-processor multiprocessor by what is called a *Brent-type Theorem*, which establishes the following bounds on the time t required to evaluate an expression with work w and span s on such a platform:

$$\max(w/p, s) \leq t \leq w/p + s.$$

The first inequality states that the running time is bounded below by the larger of w/p and s, meaning that work is done in chunks of p at a time insofar as that is possible, up to the fundamental limit imposed by the span. Intuitively, if a computation consists of a long chain of sequentially dependent calculations, then one cannot execute it faster than the span, regardless of how many processors one may have available. On the other hand, if the computation has a lot of independent chunks of work, then the best we can do is perform it in chunks of p at a time, keeping each processor busy. The second inequality says that there is a way to schedule the evaluation so that it takes no more than the sum of the same two factors, which is at most twice the stated lower bound. The main idea is to use a so-called greedy scheduler that ensures that no processor remains idle when there is work to be done, thereby maximizing progress on the work.

Bounded Arithmetic and Arrays

Fix a parameter $b > 0$, representing the number of bits in an unsigned fixed-length number. The type `int` is the type of such numbers, with the parameter b being implicit (and fixed).[5]

Bounded integer constants and operations:

$$\begin{array}{rcll} e & ::= & \overline{n} & \text{numeric literal} \\ & & e_1 \oplus e_2 & \text{addition} \\ & & e_1 \ominus e_2 & \text{cut-off subtraction} \end{array}$$

Their cost semantics (omitting subtraction for brevity):

$$\frac{0 \leq n < 2^b}{\overline{n}\ \mathsf{val}} \qquad \frac{n \geq 2^b}{\overline{n} \Uparrow [1;1]}$$

$$\frac{e_1 \Downarrow \overline{n_1}\ [w_1;s_1] \quad e_2 \Downarrow \overline{n_2}\ [w_2;s_2] \quad n = n_1 + n_2 < 2^b}{e_1 \oplus e_2 \Downarrow \overline{n}\ [w_1 + w_2 + 1; \max(s_1, s_2) + 1]}$$

$$\frac{e_1 \Downarrow \overline{n_1}\ [w_1;s_1] \quad e_2 \Downarrow \overline{n_2}\ [w_2;s_2] \quad n = n_1 + n_2 \geq 2^b}{e_1 \oplus e_2 \Uparrow [w_1 + w_2 + 1; \max(s_1, s_2) + 1]}$$

$$\frac{e_1 \Uparrow [w_1;s_2] \quad e_2 \Updownarrow [w_2;s_2]}{e_1 \oplus e_2 \Uparrow [w_1 + w_2 + 1; \max(s_1, s_2) + 1]}$$

$$\frac{e_1 \Updownarrow [w_1;s_2] \quad e_2 \Uparrow [w_2;s_2]}{e_1 \oplus e_2 \Uparrow [w_1 + w_2 + 1; \max(s_1, s_2) + 1]}$$

Each operation on `int` has constant work and span, and alway yields a value that is within range. The key is the *a priori* bound, otherwise it is impossible to perform arithmetic in constant time. An out of range computation is an error, as opposed to a more typical convention of "wrapping around" silently.

Arrays are the primitive notion from which more useful notions of sequence are implemented. An array is a collection of values indexed by an `int`, a bounded unsigned integer. Restriction of the index to `int`'s is necessary to ensure that array access takes constant work and span.

[5] It seems that one could instead consider a family of types `int`$[b]$ parameterized by the bit length, but it would be overkill for present purposes.

Array operations:

$$\begin{array}{rcll} e & ::= & \langle\!\langle v_0, \ldots, v_{n-1} \rangle\!\rangle_n & \text{array value} \\ & & \langle e_2 \mid 0 \leq x < e_1 \rangle & \text{tabulate an array} \\ & & |e| & \text{length of an array} \\ & & e_1[e_2] & \text{element of an array} \end{array}$$

The rules defining evaluation of array expressions are as follows:

$$\frac{v_0\ \mathsf{val} \quad \ldots \quad v_{n-1}\ \mathsf{val}}{\langle\!\langle v_0, \ldots, v_{n-1} \rangle\!\rangle_n\ \mathsf{val}}$$

$$\frac{e_1 \Downarrow \overline{n}\ [w; s] \quad [\overline{0}/x]e_2 \Downarrow v_0\ [w_0; s_0] \quad \ldots \quad [\overline{n-1}/x]e_2 \Downarrow v_{n-1}\ [w_{n-1}; s_{n-1}]}{\langle e_2 \mid 0 \leq x < e_1 \rangle \Downarrow \langle\!\langle v_0, \ldots, v_{n-1} \rangle\!\rangle_n\ [w + (\sum_{i=0}^{n-1} w_i) + 1; s + (\max_{i=0}^{n-1} s_i) + 1]}$$

$$\frac{e \Downarrow \langle\!\langle v_0, \ldots, v_{n-1} \rangle\!\rangle_n\ [w; s]}{|e| \Downarrow \overline{n}\ [w + 1; s + 1]}$$

$$\frac{e_1 \Downarrow \langle\!\langle v_0, \ldots, v_{n-1} \rangle\!\rangle_n\ [w_1; s_1] \quad e_2 \Downarrow \overline{k}\ [w_2; s_2] \quad k < n}{e_1[e_2] \Downarrow v_k\ [w; s]}$$

An array value is an indexed collection of values labelled with its size. Tabulation creates an array of a given size from an expression that computes the ith element as a function of i. Tabulation charges one unit of cost for creating the array.[6] Selecting an element of an array takes unit work and span, and incurs an error if the index of the element is out of bounds.

The error rules pertaining to arrays are as follows:

$$\frac{e_1 \Downarrow \overline{n}\ [w; s] \quad [\overline{0}/x]e_2 \Updownarrow\ [w_0; s_0] \quad \ldots \quad [\overline{j}/x]e_2 \Uparrow\ [w_j; s_j] \quad \ldots \quad [\overline{n-1}/x]e_2 \Updownarrow\ [w_{n-1}; s_{n-1}]}{\langle e_2 \mid 0 \leq x < e_1 \rangle \Uparrow\ [w + (\sum_{i=0}^{n-1} w_i) + 1; s + (\max_{i=0}^{n-1} s_i) + 1]}$$

$$\frac{e \Uparrow\ [w; s]}{|e| \Uparrow\ [w + 1; s + 1]}$$

$$\frac{e_1 \Downarrow \langle\!\langle v_0, \ldots, v_{n-1} \rangle\!\rangle_n\ [w_1; s_1] \quad e_2 \Downarrow \overline{k}\ [w_2; s_2] \quad k \geq n}{e_1[e_2] \Uparrow\ [w_1 + w_2 + 1; \max(s_1, s_2) + 1]}$$

$$\frac{e_1 \Uparrow\ [w_1; s_1] \quad e_2 \Updownarrow\ [w_2; s_2]}{e_1[e_2] \Uparrow\ [w_1 + w_2 + 1; \max(s_1, s_2) + 1]}$$

[6] This choice is justified by the restriction of the size to an `int`, the type of bounded unsigned integers.

$$\frac{e_1 \Updownarrow\ [w_1; s_1] \quad e_2 \Uparrow\ [w_2; s_2]}{e_1[e_2] \Uparrow\ [w_1 + w_2 + 1; \max(s_1, s_2) + 1]}$$

If any component of a tabulated array incurs an error, then the tabulate also incurs an error. As with n-tuples, the cost incurs that of evaluating all other components, because of parallelism. Accessing a component outside of the range of an array incurs an error. In all other cases errors are propagated from the component expressions.

The subarray, or restriction, operation, $e[e_1..e_2]$, computes the array of length $e_2 - e_1$ given by the elements of e from e_1 up to e_2. It is easily definable from tabulate:[7]

$$\langle e[e_1 \oplus x] \mid 0 \leq x < e_2 \rangle.$$

However, this formulation takes linear work and constant span. By making restriction a primitive operation we can implement it with constant work and span. The main idea is to generalize array values to have the form

$$\langle\!\langle v_0, \ldots, v_{m-1} \rangle\!\rangle_n^k$$

where $k \oplus n \leq m$. This value represents the length-n segment starting at index k of the underlying length-m array. The evaluation rules for the array primitives all change slightly to accommodate this more general form of value, but without changing their costs. The restriction operation can then be specified as having unit work and span. The rules are as follows:

$$\frac{v_0\ \mathsf{val} \quad \ldots \quad v_{m-1}\ \mathsf{val} \quad n \oplus k \leq m}{\langle\!\langle v_0, \ldots, v_{m-1} \rangle\!\rangle_n^k\ \mathsf{val}}$$

$$\frac{e_1 \Downarrow \overline{n}\ [w; s] \quad [\overline{0}/x]e_2 \Downarrow v_0\ [w_0; s_0] \quad \ldots \quad [\overline{n-1}/x]e_2 \Downarrow v_{n-1}\ [w_{n-1}; s_{n-1}]}{\langle e_2 \mid 0 \leq x < e_1 \rangle \Downarrow \langle\!\langle v_0, \ldots, v_{n-1} \rangle\!\rangle_n^0\ [w + (\sum_{i=0}^{n-1} w_i) + 1; s + (\max_{i=0}^{n-1} s_i) + 1]}$$

$$\frac{e \Downarrow \langle\!\langle v_0, \ldots, v_{m-1} \rangle\!\rangle_n^k\ [w; s]}{|e| \Downarrow \overline{n}\ [w + 1; s + 1]}$$

$$\frac{e_1 \Downarrow \langle\!\langle v_0, \ldots, v_{m-1} \rangle\!\rangle_n^k\ [w_1; s_1] \quad e_2 \Downarrow \overline{i}\ [w_2; s_2] \quad i < n}{e_1[e_2] \Downarrow v_{k \oplus i}\ [w; s]}$$

$$\frac{e \Downarrow \langle\!\langle v_0, \ldots, v_{m-1} \rangle\!\rangle_n^k\ [w; s] \quad e_1 \Downarrow \overline{i}\ [w_1; s_1] \quad e_2 \Downarrow \overline{j}\ [w_2; s_2] \quad i \leq j < n}{e[e_1..e_2] \Downarrow \langle\!\langle v_0, \ldots, v_{m-1} \rangle\!\rangle_{j \ominus i}^{k \oplus i}\ [w + w_1 + w_2 + 1; \max(s, s_1, s_2) + 1]}$$

[7] This formulation ignores checking for bounds errors, which may be easily added.

$$\frac{e \Downarrow \langle\!\langle v_0, \ldots, v_{m-1} \rangle\!\rangle_n^k \ [w; s] \quad e_1 \Downarrow \bar{i} \ [w_1; s_1] \quad e_2 \Downarrow \bar{j} \ [w_2; s_2] \quad j \geq n}{e[e_1..e_2] \Uparrow \ [w + w_1 + w_2 + 1; \max(s, s_1, s_2) + 1]}$$

The other error rules remain essentially as before, taking into account the new form of array value, and, as before, errors propagate through restriction operations.

The operation $e_1 \backslash e_2$, called injection, or multiple update, modifies an array, e_1, by a given sequence of index-value pairs, e_2. If a given index is updated more than once by e_2, the *rightmost* one determines the value at that index. Any out of bounds indices incur an error at run-time. Multiple update may be easily reduced to an iterated usage of a single-update primitive, which itself may be defined using tabulate with work proportional to the size of the array, and constant span. The work of the multiple update is then proportional to the product of the sizes of the given arrays, and the span is proportional to the number of updates. An alternative is to take multiple update as a primitive, with work proportional to the number of updates and constant span.[8]

[8] Additive work and linear span is achievable using only benign effects; constant span relies on machine support.