Towards An Operational Semantics for a Parallel Non-strict Functional Language

Jon G. Hall[1], Clem Baker-Finch[2], Phil Trinder[3], and David J. King[1]

[1] Faculty of Maths and Computing, The Open University
Milton Keynes MK7 6AA
{J.G.Hall, D.J.King}@open.ac.uk
[2] Department of Computer Science, Australian National University
Clem.Baker-Finch@cs.anu.edu.au
[3] Department of Computing and Electrical Engineering
Heriot-Watt University, Riccarton, Edinburgh EH14 4AS
trinder@cee.hw.ac.uk

Abstract. Parallel programs must describe both computation and coordination, i.e. *what* to compute and *how* to organize the computation. In functional languages equational reasoning is often used to reason about computation. In contrast, there have been many different coordination constructs for functional languages, and far less work on reasoning about coordination.
We present an initial semantics for GpH, a small extension of the Haskell language, that allows us to reason about coordination. In particular we can reason about *work*, *average parallelism* and *runtime*. The semantics captures the notions of limited (physical) resources, the preservation of sharing, and speculative evaluation. We show a consistency result with Launchbury's well-known lazy semantics.

1 Introduction

One of the advantages of declarative languages is that it is relatively easy to reason about the values computed by programs, this being attributable to their preservation of referential transparency. Indeed, within the functional programming community there is a strong tradition of reasoning to transform, derive, and prove properties of programs.

Parallel programs must describe both computation and coordination, i.e. *what* to compute, and *how* to arrange the computation in parallel. In adding coordination constructs, many parallel functional languages are able to preserve the referential transparency of the computation language [NAH93,BCH⁺93,FMS⁺95,NSvEP91] and [BLOMP97] so that standard equational reasoning techniques continue to be applicable to the values computed. Being able to reason about coordination is, however, dependent on the form its specification takes: for languages in which coordination is entirely implicit [NAH93,BCH⁺93,FMS⁺95,MS95], or in which it is specified as annotations [KG89,NSvEP91], reasoning about the coordination at the language level is not possible. It is only for languages that make the specification of coordination explicit [BLOMP97,CGKL98,Hal98] and [MH95], that such reasoning is possible at the language level.

Our language, Glasgow Parallel Haskell (GpH) [THJ$^+$96] is explicit about a few crucial aspects of the coordination, and so we would like to develop a semantics for reasoning about it. GpH is a modest extension to Haskell [ABB$^+$97], adding only two coordination primitives: parallel and sequential composition, denoted **par** and **seq** respectively. By abstracting from the primitives using higher-order polymorphic functions it is possible to cleanly separate computation from coordination, and the abstractions are called *evaluation strategies* [THLJ98].

1.1 Motivation: reasoning about coordination

The computational meaning of **par** and **seq** is captured by the following equations; both are projections onto their second argument, but only **seq** is strict in its first argument.

$$
\begin{aligned}
e_0 \ \mathbf{par}\ e_1 &= e_1 \\
e_0 \ \mathbf{seq}\ e_1 &= \begin{cases} \bot & \text{if } e_0 = \bot \\ e_1 & \text{otherwise} \end{cases}
\end{aligned}
$$

The coordination, or operational behaviour, of **seq** is for the arguments to be evaluated in sequence: the first to weak head normal form (whnf) before the second. The coordination behaviour of **par** is for the arguments to be evaluated in parallel, potentially: the first argument is marked as a candidate for parallel evaluation by a new thread, but but this will only occur if there is a free processor.

Figure 1 shows some examples of obvious value-equivalences between terms involving the coordination primitives. We wish to be able to investigate the status of such equivalences with respect to coordination and, in particular, work, average parallelism and runtime ([EZL89], see the discussion in Section 5) and the transition system we present here is a first step towards this goal. Ultimately, however, in this context the usefulness of a semantics for GpH will be measured by how it allows one to reason about programs which involve evaluation strategies, rather than just the **par** and **seq** primitives.

1.2 Related Work

The initial motivation for our semantics comes from John Launchbury's natural semantics of lazy evaluation [Lau93]. Readers familiar with that work will recognise a number of features here, including the normalisation process and the explicit heap of uniquely identified closures.

(I) Associativity of **seq**:

$$e_0 \textbf{ seq } (e_1 \textbf{ seq } e_2) = (e_0 \textbf{ seq } e_1) \textbf{ seq } e_2$$

(II) Idempotence of e_0 **par**:

$$e_0 \textbf{ par } e_1 = e_0 \textbf{ par } e_0 \textbf{ par } e_1$$

(III) Distribution of **seq** over **par**

$$e_0 \textbf{ seq } (e_1 \textbf{ par } e_2) = (e_0 \textbf{ seq } e_1) \textbf{ par } (e_0 \textbf{ seq } e_2)$$

Fig. 1. Some obvious value-equivalences between terms involving the coordination primitives.

Launchbury presents an evaluation semantics (big-step SOS) but it is generally agreed that such an approach is inappropriate for describing parallelism [Hen90]. Hence we build a computational semantics (small-step SOS) over structures similar to those introduced by Launchbury. The semantics is shown, in Section 4, to be consistent with that of Launchbury on the overlap. (Moreover, as Launchbury's semantics is shown to be consistent with Abramsky denotational semantics of the Lazy Lambda Calculus of [Abr90], by transitivity, we have a similar consistency result.)

A distinguishing feature of the approach taken in this paper is that the computational steps are explicitly parallel, being based on lockstep synchrony ([Mil89]). This contrasts with the common approach exemplified by many process algebras, of *representing* parallelism by interleaving computational steps [Hen88], but allows a more natural consideration of resource usage.

The computational model we introduce for parallel non-strict evaluation is a direct extension of that of the Launchbury semantics, and we characterise their relationship in Section 4. In particular, the semantics captures the preservation of sharing but augments Launchbury's semantics with the notions of limited (physical) resources. We note that the current semantics does not model GpH, but models a language with speculative evaluation, a point which is discussed in Section 5.

1.3 Structure of the paper

The paper is organised as follows: Section 2 describes the extended form of normalisation that the introduction of **par** and **seq** require, together with the rules which comprise the transition system for our parallel language. Section 3 gives example derivations illustrating the semantics, and in particular how it models limited resources. Section 4 contains a preliminary exploration of the relationship between this work and [Lau93], and of other properties of the semantics.

We end with a critique of the work which motivates our future work, together with conclusions and an outlook.

2 A transition system for the parallel language

The semantics we define is a small step Plotkin-style Structured Operational Semantics [Plo81]. In the semantics steps correspond to single (lockstep) reductions. This contrasts with the natural semantics defined in [Lau93] in which steps correspond to full reductions to whnf. The benefits of a natural semantics are well stated in [Lau93] with the hope there being that the high level of abstraction of a natural semantics will provide for the study of a broad spread of implementations and facilitate the proofs of properties.

Our semantic basis of a lockstep semantics could, from this point of view, be seen as a retrograde step. However, the inclusion of parallelism into the semantics appears to require it: even though we would like to be able to express parallelism between full reductions to whnf there are coordination requirements between thread creation and destruction which require consideration of behaviours at the level of single reductions.

2.1 Normalisation

The transition system is based on a lambda calculus extended with recursive **let**s, and **seq** and **par**, which we normalise to the following restricted syntax. The syntax resembles closely that of [Lau93] in its simplicity, and shares the distinguishing features that all bound variables are distinct so that scope is irrelevant. However, whereas in [Lau93] applications are of the form *an expression applied to a variable*, for us applications are only allowed when they are of the form *a variable applied to a variable.* Although motivation for the restrictions is shared between the two approaches—it removes the necessity to generate new closure sites when **let**s are moved into the 'heap'—we have a different form of judgment to that of [Lau93] and must to forego the 'luxury' of an unnamed site. Because of this a function in an application must be transformable to an explicitly named site as well.

The restricted syntax for our extended lambda calculus is:

$$
\begin{array}{rl}
w, x, y, z \in \mathit{Var} & \\
e \in \mathit{Exp} ::= & \lambda\, x . e \\
\mid & y\ x \\
\mid & x \\
\mid & \mathbf{let}\ x_1 = e_1, \ldots, x_n = e_n\ \mathbf{in}\ e \\
\mid & x\ \mathbf{par}\ e \\
\mid & x\ \mathbf{seq}\ e
\end{array}
$$

(Representatives of syntactic categories will be decorated as and when necessary.)

The reader will note that **seq** and **par** are allowed expressions as their second argument; this is because of their projective nature ($e_1 \,\mathbf{par}\, e_2 = e_2$ and $e_1 \neq \perp \Rightarrow e_1 \,\mathbf{seq}\, e_2 = e_2$), and the fact that they will be an explicitly named closure in the semantics. There projective nature implies that it is safe to reuse this name for their second argument.

Transforming an arbitrary term to the above syntax is done through the process of normalisation which is defined next. The first stage of normalisation is to produce $\hat{e}$ from e. $\hat{e}$ is e to which α-conversion has been applied renaming all bound variables to 'fresh' variable names. Although it also applies to lambda terms the main use of this step is so that, when a **let** expression is moved to the 'heap', there will be no clashes in variable names.

The second stage in the normalisation of a term is to reduce it to the restricted syntax. For a term e we define:

$$
\begin{array}{ll}
(\lambda\, x.e)^* & = \lambda\, x.(e^*) \\
x^* & = x \\
\multicolumn{2}{l}{(\mathbf{let}\ x_1 = e_1, \ldots, x_n = e_n\ \mathbf{in}\ e)^*} \\
 & = \mathbf{let}\ x_1 = (e_1^*), \ldots, x_n = (e_n^*)\ \mathbf{in}\ (e^*) \\
(e_1\ e_2)^* & = \begin{cases} e_1\ e_2 & \text{if } e_1 \text{ and } e_2 \text{ are variables} \\ \mathbf{let}\ y = (e_2^*)\ \mathbf{in}\ (e_1\ y) & \text{if } e_1 \text{ is a variable} \\ \mathbf{let}\ x = (e_1^*)\ \mathbf{in}\ (x\ e_2) & \text{if } e_2 \text{ is a variable} \\ \mathbf{let}\ x = (e_1^*), y = (e_2^*)\ \mathbf{in}\ (x\ y) & \text{otherwise} \end{cases} \\
(e_1\ \mathbf{par}\ e_2)^* & = \begin{cases} e_1\ \mathbf{par}\ (e_2^*) & \text{if } e_1 \text{ is a variable} \\ \mathbf{let}\ x = (e_1^*)\ \mathbf{in}\ (x\ \mathbf{par}\ (e_2^*)) & \text{otherwise} \end{cases} \\
(e_1\ \mathbf{seq}\ e_2)^* & = \begin{cases} e_1\ \mathbf{seq}\ (e_2^*) & \text{if } e_1 \text{ is a variable} \\ \mathbf{let}\ x = (e_1^*)\ \mathbf{in}\ (x\ \mathbf{seq}\ (e_2^*)) & \text{otherwise} \end{cases}
\end{array}
$$

where each introduced variable is 'fresh'.

Other than for the case of application (as described above), normalisation here and in [Lau93], are the same for the core language. The extra cases of **par** and **seq** provide for the expanded syntax of the strategic extensions to the language.

2.2 Transition system

The rules which define the lockstep semantics are presented in Figure 2, and their detail will be described in Section 2.4. Elements in the figure (and, more widely, the paper) follow naming conventions thus:

$$
\begin{array}{rl}
\Delta, \Gamma \in \mathit{Heap} & = \ \mathit{Var} \twoheadrightarrow \mathit{Exp} ::= \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\} \\
v \in \mathit{Val} & ::= \lambda\, x.e
\end{array}
$$

A heap is a partial function from variables to expressions. A value is an expression in whnf.

Sequential Rules

$$\begin{array}{ll}\Delta : (y \mapsto \textbf{let } x_1 = e_1, \ldots, x_n = e_n \textbf{ in } e) & \\ \qquad\qquad \longrightarrow (\Delta, x_1 \mapsto e_1, \ldots, x_n \mapsto e_n) : (y \mapsto e) & \textit{Let}\end{array}$$

$$(\Delta, x \mapsto \lambda w.e) : (z \mapsto x\ y) \longrightarrow (\Delta, x \mapsto \lambda w.e) : (z \mapsto e[y/w]) \qquad \textit{Application}$$

$$(\Delta, x \mapsto v) : (y \mapsto x) \longrightarrow (\Delta, x \mapsto v) : (y \mapsto \hat{v}) \qquad \textit{Variable}$$

$$(\Delta, x \mapsto v) : (z \mapsto x \textbf{ seq } e) \longrightarrow (\Delta, x \mapsto v) : (z \mapsto e) \qquad \textit{Sequence}$$

$$\Delta : (z \mapsto x \textbf{ par } e') \longrightarrow \Delta : (z \mapsto e') \qquad \textit{Parallel1}$$

$$(\Delta, x \mapsto e) : (z \mapsto x \textbf{ par } e') \longrightarrow \Delta : (x \mapsto e, z \mapsto e') \qquad \textit{Parallel2}$$

Parallel Rules

$$\frac{\Delta : \tau_i \longrightarrow \Delta_i : \tau_i', \quad 1 \leq i \leq n_{red} \qquad n_{red} + m_a \leq \textbf{max}}{\Delta : \Gamma \longrightarrow \Gamma_d \cup \bigcup \Delta_i \setminus \Delta_a : \Delta_a \cup \bigcup \tau_i'} \qquad \textit{Product}$$

Fig. 2. The small-step transition system

A judgment in the transition system is of the form:

$$\Delta : \Gamma \longrightarrow \Delta' : \Gamma'$$

which should be read as

> the *live* bindings Γ in the context of the *dead* bindings Δ in one step become the *live* bindings Γ' in the context of the *dead* bindings Δ'.

As we shall see, the concept of 'one-step' does not restrict us to a single change in the live heap as the 'one-step' is a single *lock*step, and may consist of any number of single steps (under the proviso that sufficient resources are available).

2.3 The parallel abstract machine

The transition system defines an abstract machine for the interpretation of terms in our parallel functional language. The abstract machine has explicit initial and terminal states so that we can begin and end computations from known points.

The *initial configuration* of the abstract machine is:

$$\varnothing : \{main \mapsto e\}$$

where *main* is the thread that drives the computation. e is likely to be a *let* expression. Note that, as we explicitly name all closures, *main* is explicit in the initial configuration.

The *terminal configuration* of a program is *not* that in which all calculation has ceased—such a scheme would allow non-terminating computations uninvolved in the determination of the value of *main* to prevent termination—rather that in which a (whnf) value has been returned for the *main* thread:

$$\Delta : (\Gamma, main \mapsto v)$$

The rules of the system are described next.

2.4 Two rule forms

There are two forms of rules in the system: sequential and parallel. The sequential rules apply to what we might call (in the spirit of [Mil89, Pg. 196]) *particulate actions* or *particles*, i.e., the building blocks from which locksteps are built. Particles are characterised by the singleton nature of the live heap *on the lhs of a judgment*[1] (that before the :).

[1] Not necessarily the rhs of a judgment.

Parallel rules, of which *Product* is the only example, build locksteps from particles, in a way similar to the **Prod** rule of [Mil89], i.e., if, in a particular state, particles can individually perform steps $s_1, \ldots, s_n$, then Product will combine these into the single lockstep $\{s_1, \ldots, s_n\}$. Because of this, applications of *Product* are justified through individual applications of the sequential rules to the constituent particles.

In addition, *Product* allows live, but whnf, terms to be switched out and dead, non-whnf, terms to have resources assigned to them, which is not the case for Milner's **Prod** rule.

Application, Variable, Let These rules allow progression in the computation associated with a single named closure in the heap.

The *Let* rule transforms the binding list of a **let** expression into named closures in the dead heap. We recall that the first normalisation step renames all bound variables to 'fresh' variables, so that this transformation will not produce name clashes.

For *Application* we assume that the function (bound to variable x in the rule) has already been reduced to whnf and appears in the dead environment. The transition associated with *Application* is to perform the β-reduction.

The *Variable* rule allows a variable whose binding is already whnf in the dead environment to be replaced by its whnf value.

Parallel and Sequence The *Sequence* rule states that if we have completed the computation of the first argument of a **seq** (i.e., it is in whnf), we should make a start on the second argument. That we require a completed reduction to whnf term for the first argument before starting the second provides the source of the strictness of **seq** in its first argument.

The *Parallel*2 rule assigns a resource to the calculation of its first and second arguments. We assume that the first argument was dead. This rule will be allowed in the justification of a *Product* step if there are sufficient resources to be applied to both arguments. The *Parallel*1 rule applies if there are insufficient resources: it discards its first operand. The determination of which rule to apply is made in the *Product* rule, described next.

Product The *Product* rule models our lockstep semantics of parallelism, and its resource consciousness.

The rule requires the following assumptions on the form of the heaps; that either:

- a live term can being reduced using one of the *Application*, *Variable*, *Let*, *Sequence* or one of the two parallel rules (which, in the case of the *Parallel*2, will introduce a thread into the live heap from the dead heap);

– an *irreducible* expression can be moved from the live heap to the dead heap. (In this context, by irreducible we mean that the expression is already in whnf, or is one of $x\ y$ or $x\ \mathbf{seq}\ e$ where x (or the expression bound to x) is not in whnf in the dead environment.)
– a non-whnf expression can be assigned a resource, and so moved from the dead heap to the live heap; or
– a dead thread can remain in the dead heap.

To keep the rule in Figure 2 'page sized', we have used the following notation:

– $\Delta = \Delta_a \cup \Delta_d$ with $\Delta_a = \{y_1 \mapsto e_1, \ldots, y_{m_a} \mapsto e_{m_a}\}$ being those dead threads which become active either through the switching in of a dead thread (in which case e_i is not a whnf expression), or because of an application of the *Parallel*2 rule, and $\Delta_d = \{y_{m_a+1} \mapsto e_{m_a+1}, \ldots, y_m \mapsto e_m\}$ being those dead threads that remain dead (in which case e_i may or may not be a whnf expression).
– $\Gamma = \Gamma_{red} \cup \Gamma_d$ with $\Gamma_{red} = \{\tau_1, \ldots, \tau_{n_{red}}\}$, $\tau_i = x_i \mapsto e_i$, being those active (and hence non-whnf) threads that reduce, and $\Gamma_d = \{x_{n_{red}+1} \mapsto v_{n_{red}+1}, \ldots, x_n \mapsto v_n\}$ being those alive, but irreducible, threads that become dead. Δ_i is the change wrought on the dead environment through the reduction of τ_i to τ_i'.

The indices of the variables provide as to their behaviour. Those in the live heap between 1 and n_{red} will reduce. Those in the live heap which will migrate to the dead heap can be found between indices $n_{red}+1$ and n (this covers all indices: a thread is either reducible and reduce in the next step or is in whnf and so have its assigned resource recovered). Those in the dead heap which migrate to the live heap will be found from indices 1 and n_a.

The only other component of the rule is the proviso involving **max** which ensures that the movement of dead to live and live to dead does not exceed the level of resources available.

2.5 Building Parallel Behaviours

Because of the spilt between sequential and parallel rules, parallel behaviours are built solely from applications of the *Product* rule with the other rules only being used in the justification of their application. In derivations of lockstep sequences this should mean that all steps have the name *Product*, which is rather unhelpful. One might consider leaving individual steps in a derivation unlabelled; but, at least in small derivations, the reading of a derivation is facilitated by annotating *Product* rule application with its sequential justifying steps. So, for instance, the step

$$\begin{array}{l} \varnothing : \{main \mapsto \mathbf{let}\ f = \lambda\, x.x, g = \lambda\, x.x, gx = g\ x\ \ \mathbf{in}\ \ gx\ \mathbf{par}\, f\ \ gx\} \\ \longrightarrow \{f \mapsto \lambda\, x.x, g \mapsto \lambda\, x.x, gx \mapsto g\ \ x\} : \{main \mapsto gx\ \mathbf{par}\, f\ \ gx\}\ \mathit{Product} \end{array}$$

...

in which a *Let* step has been used to justify the *Product* will be written

$$\begin{array}{ll} \varnothing : \{main \mapsto \textbf{let}\ f = \lambda\, x.x, g = \lambda\, x.x, gx = g\ x\ \textbf{ in }\ gx\ \textbf{par}\ f\ gx\} & \\ \longrightarrow \{f \mapsto \lambda\, x.x, g \mapsto \lambda\, x.x, gx \mapsto g\ x\} : \{main \mapsto gx\ \textbf{par}\ f\ gx\} & \mathit{Let} \\ \ldots & \end{array}$$

When two or more sequential rules are used in the justification of a single application we will write them as a comma separated list. When the only steps involved in an application of *Product* are the switching-in or switching-out of threads, we will annotate and application with *Product*.

3 Example

As an example of the transition system in action, we present the derivation of the term

$$main = \textbf{let}\ f = \lambda\, x.x, g = \lambda\, x.x, gx = g\ x\ \textbf{ in }\ gx\ \textbf{par}\ f\ gx$$

in two situations:

1. when $\mathbf{max} = 1$, with a single processor architecture;
2. when $\mathbf{max} = 2$, with a dual-processor architecture.

The details of the term are not particularly interesting, other than for the fact that *main* contains a **par** construct. This fact should distinguish the two situations.

In the case when $\mathbf{max} = 1$ we have that:

$$\begin{array}{ll} \varnothing : \{main \mapsto \textbf{let}\ f = \lambda\, x.x, g = \lambda\, x.x, gx = g\ x\ \textbf{ in }\ gx\ \textbf{par}\ f\ gx\} & \\ \longrightarrow \{f \mapsto \lambda\, x.x, g \mapsto \lambda\, x.x, gx \mapsto g\ x\} : \{main \mapsto gx\ \textbf{par}\ f\ gx\} & \mathit{Let} \\ \longrightarrow \{f \mapsto \lambda\, x.x, g \mapsto \lambda\, x.x, gx \mapsto g\ x\} : \{main \mapsto f\ gx\} & \mathit{Parallel1}(\dagger) \\ \longrightarrow \{f \mapsto \lambda\, x.x, g \mapsto \lambda\, x.x, gx \mapsto g\ x\} : \{main \mapsto gx\} & \mathit{Application} \\ \longrightarrow \{f \mapsto \lambda\, x.x, g \mapsto \lambda\, x.x, gx \mapsto g\ x\} : \{main \mapsto g\ x\} & \mathit{Variable} \\ \longrightarrow \{f \mapsto \lambda\, x.x, g \mapsto \lambda\, x.x, gx \mapsto x\} : \{main \mapsto x\} & \mathit{Application} \end{array}$$

For comparison, in the case when $\mathbf{max} = 2$ we have, rather, that:

$$\begin{array}{ll} \varnothing : \{main \mapsto \textbf{let}\ f = \lambda\, x.x, g = \lambda\, x.x, gx = g\ x\ \textbf{ in }\ gx\ \textbf{par}\ f\ gx\} & \\ \longrightarrow \{f \mapsto \lambda\, x.x, g \mapsto \lambda\, x.x, gx \mapsto g\ x\} : \{main \mapsto gx\ \textbf{par}\ f\ gx\} & \mathit{Let} \\ \longrightarrow \{f \mapsto \lambda\, x.x, g \mapsto \lambda\, x.x\} : \{gx \mapsto g\ x, main \mapsto f\ gx\} & \mathit{Parallel2}(\dagger) \\ \longrightarrow \{f \mapsto \lambda\, x.x, g \mapsto \lambda\, x.x\} : \{gx \mapsto x, main \mapsto gx\} & \mathit{Application}, \\ & \quad \mathit{Application} \\ \longrightarrow \{f \mapsto \lambda\, x.x, g \mapsto \lambda\, x.x, gx \mapsto x, main \mapsto gx\} : \varnothing & \mathit{Product}(*) \\ \longrightarrow \{f \mapsto \lambda\, x.x, g \mapsto \lambda\, x.x, gx \mapsto x\} : \{main \mapsto gx\} & \mathit{Product}(**) \\ \longrightarrow \{f \mapsto \lambda\, x.x, g \mapsto \lambda\, x.x, gx \mapsto x\} : \{main \mapsto x\} & \mathit{Variable} \end{array}$$

Notes:

1. In the case of the second derivation we provide the justification of the first *Product* step (labelled *Application*, *Application*) involving a lockstep between two *Application* applications: one on f in gx and one on g in $main$. The justification steps are:

$$\{f \mapsto \lambda\, x.x, g \mapsto \lambda\, x.x\} : \{gx \mapsto g\ x\}$$
$$\longrightarrow \{f \mapsto \lambda\, x.x, g \mapsto \lambda\, x.x\} : \{gx \mapsto x\}\ \mathit{Application}$$

$$\{f \mapsto \lambda\, x.x, g \mapsto \lambda\, x.x\} : \{main \mapsto f\ gx\}$$
$$\longrightarrow \{f \mapsto \lambda\, x.x, g \mapsto \lambda\, x.x\} : \{main \mapsto gx\}\ \mathit{Application}$$

2. In the second derivation, in the second *Product* application (labelled $*$) we have set $\Gamma_d = \{gx \mapsto x, main \mapsto gx\}$, and used *Product* to switch out all irreducible terms.
3. In the third application (labelled $**$) we have set $\Delta_a = \{main \mapsto gx\}$, and used *Product* to switch in $main$.
4. The limitation of resources to 1 in the first case prevents the application of the *Parallel*2 rule at the step marked †; if *Parallel*2 were to be applied then $n_{red} + m_a = 2 > \mathbf{max}$. $\mathbf{max} = 2$ in the second derivation so that *Parallel*2 may be applied in this case.

3.1 On execution times

Clearly, the length of a computation should be proportional to the number of reductions involved in its derivation, so that the measure of execution time is related to the length of a derivation. This is reflected, to at least some degree, in the presented semantics: *Product* in a single step can combine many steps together; we should expect therefore that, at least in the general case, executions involving *Product* will be shorter. However, even though the availability of resources will often bring with it speed-ups in a computation, there are administrative costs associated with assigning a resource so that the use of parallelism may produce slow-downs instead. This corresponds to our real-world experiences of parallelism.

The lockstep nature of the rule assumes much of the timing of 'swapping-in' and 'swapping-out' operations: *viz*, that they take the same time as each other and, moreover, that they both take the same time as an ordinary step. This might be an unrealistic assumption, but it simplifies the rule greatly.

4 Semantic Propriety

In this section we show consistency with the semantics of [Lau93], and explore a little the relationship between parallel and sequential computations. The intention is the consistency result is to show consistency with the semantics of [Abr90], at least for a subset of the semantics.

To compare our semantics with that of [Lau93] we derive rules which look like the reduction steps of that semantics from our small step rules and *vice-versa*. The translation is quite natural and is done by induction on the length of a derivation. The only real difficulty with the consistency proof is that our semantics includes parallelism. We ameliorate this problem by initially restricting ourselves to the application of rules in which $\mathbf{max} = 1$, i.e., there is a single resource for computation (which means that, for any live heap Γ of the computation, $|\Gamma| = 1$) and then showing that, in almost all cases, parallel computations can be made equivalent to sequential ones. The implications of this will be clear to the reader on inspection of the transition system's rules in that the system begins to look very much like [Lau93], any minor differences being only that there is a named single binding on the right of the colon.

The case for which a completely general result fails is dues to the following: although one might expect that parallelism shouldn't add anything to the values computed by a computation, there are terms for which parallelism prevents termination. An example of such a term is

$$\mathbf{let}\ x = x\ \mathbf{par}\ 5\ \mathbf{in}\ x$$

when there are infinite resources; (after application of the *Let* rule) the term tries to use all resources — grabbing them one at a time using *Parallel*2 — and hence will not terminate. Note however that, even for this term, as long as resources are finite the problem disappears: the expression stops trying to grab another resource when only one remains and gets on with the computation, eventually returning the value 5.

4.1 Consistency with Launchbury's semantics

Manipulating a system in which bindings appear on both sides means a little extra work in our demonstration of the relationship; in particular we should show that the manipulations of such unbound thunks can be made even when bindings are involved.

To do this we prove two auxiliary properties, and derive two useful rules for *activating* and *deactivating* thunks. The first lemma states that the binding of an unused variable in the dead heap does not invalidate a computation:

Lemma 1. *Suppose that*

$$\Delta : (\Gamma, x \mapsto e) \longrightarrow^* \Delta' : (\Gamma', x \mapsto e')$$

then

$$(\Delta, w \mapsto x) : \Gamma x \mapsto e \longrightarrow^* (\Delta', w \mapsto x) : (\Gamma', x \mapsto e')$$

whenever $w \notin dom\,\Delta \cup dom\,\Gamma \cup \mathbf{FV}(e)$.

Proof As w is not referenced in the existing heaps, it does not destroy the property of unique naming. The result follows. $\square$

The second lemma is slightly more complex, and states that we may arbitrarily rename binding variables, as long as they are not referenced, without invalidating a computation.

Lemma 2. *Suppose that*

$$\Delta : (\Gamma, x \mapsto e) \longrightarrow^* \Delta' : (\Gamma', x \mapsto v)$$

with v in whnf and $x \notin dom\,\Delta \cup dom\,\Gamma$. Then

$$\Delta : (\Gamma, temp \mapsto e \longrightarrow^*)\Delta' : (\Gamma', temp \mapsto v)$$

where $temp \notin dom\,\Delta \cup dom\,\Gamma$.

Proof From the assumptions we claim that e cannot depend on the value of x in reducing to v. For, suppose that it does. Then x depends directly upon its own value, and so should have the denotation $\perp$. But then e will never reach whnf in contradiction of the statement.

Hence e cannot depend directly on the value of x in the reduction. Removing x from the environment and replacing it with *temp* will not invalidate the reduction and we have the result. □

The two derived rules we introduce allow one to 'swap out' an active process, and 'swap in' an inactive process.

Lemma 3. *a. For all Δ and v*

$$\Delta : x \mapsto v \longrightarrow \Delta \cup x \mapsto v : \varnothing \quad .$$

b. For all Δ and v, when $\mathbf{max} \geq 1$,

$$(\Delta, x \mapsto e) : \varnothing \longrightarrow \Delta : x \mapsto e \quad .$$

Proof a. Set $\Delta_a = \varnothing$, $\Delta_d = \Delta$, $\Gamma_{red} = \varnothing$, $\Gamma_d = \{x \mapsto v\}$ in the *Product* rule.

b. If $\mathbf{max} \geq 1$ we may set $\Delta_a = \{x \mapsto e\}$, $\Delta_d = \Delta \setminus \Delta_a$, $\Gamma_{red} = \varnothing$, $\Gamma_d = \varnothing$ in the *Product* rule for the result. □

Theorem 1. *Let $\longrightarrow_A$ be the transition step relation based on Let, Application, Variable, In, and Out. Let $\Downarrow$ be the natural semantics relation of [Lau93].*

Then

1. $\Downarrow \subseteq \longrightarrow_A^*$.
2. *suppose $[\Gamma] \longrightarrow_A^* [\Delta]$. Then for all x such that $(x \mapsto e) \in [\Gamma]$ and $(x \mapsto v) \in [\Delta]$ for some value v, there exists some Θ such that $\Gamma : x \Downarrow \Theta : v$.*

The second property in the statement is clearly the strongest that exists as, under $\longrightarrow_A^*$, we can reduce to non-whnf expressions whereas Launchbury's reductions always end in whnf expressions.

Proof

1. We will derive each of the rules of [Lau93] from our transition step semantics. For *Lambda* and *Let* the result follows from a simple application of the rule with the same name in our system.
   Variable: Assume that

   $$\Gamma : temp \mapsto e \longrightarrow_A^* \Delta : temp \mapsto v$$

   where *temp* is as in the Launchbury convention. Then

   $$\begin{array}{ll} & (\Gamma, x \mapsto e) : temp' \mapsto x \\ \longrightarrow_{Out/In} & (\Gamma, temp' \mapsto x) : x \mapsto e \\ \longrightarrow_{*} & (\Delta, temp' \mapsto x) : x \mapsto z \\ \longrightarrow_{Out/In} & (\Delta, x \mapsto z) : temp' \mapsto x \\ \longrightarrow_{Var} & (\Delta, x \mapsto z) : temp' \mapsto \hat{z} \end{array}$$

   Application: Assume that

   $$\Gamma : temp \mapsto e \longrightarrow_A^* \Delta : temp \mapsto \lambda\, y.e'$$

   and that

   $$\Delta : temp' \mapsto e'[x/y] \longrightarrow_A^* \Theta : temp' \mapsto z \quad .$$

   Then

   $$\begin{array}{ll} & (\Gamma, y \mapsto e) : temp \mapsto y\ x \\ \longrightarrow_{Out/In} & (\Gamma, temp \mapsto y\ x) : y \mapsto e \\ \longrightarrow_A^* & (\Delta, temp \mapsto y\ x) : y \mapsto \lambda\, y.e' \\ \longrightarrow_{Out/In} & (\Delta, y \mapsto \lambda\, y.e') : temp \mapsto y\ x \\ \longrightarrow_{Appl} & (\Delta, y \mapsto \lambda\, y.e') : temp \mapsto e'[x/y] \\ \longrightarrow_A^* & (\Theta, y \mapsto \lambda\, y.e') : temp \mapsto z \end{array}$$

   as required.
2. We first introduce some auxiliary notation: suppose Γ is a heap $\{x_i \mapsto e_i\}_{i=1}^n$. We write $[\Gamma]$ to indicate an arbitrary configuration $\{x_1 \mapsto e_1, \ldots x_{j-1} \mapsto e_{j-1}, x_{j+1} \mapsto e_{j+1}, \ldots x_n \mapsto e_n\} : x_j \mapsto e_j$ for some $1 \leq j \leq n$. This helps to filter out irrelevant detail in the proof. By the *Out/In* rule, all such $[\Gamma]$ are in a sense equivalent.
   The proof of this part is by induction on the length of the derivation. Consider the last transition step in each case.
   Let: Suppose

   $$[\Gamma] \longrightarrow_A^* \Gamma' : (x \mapsto \textbf{let}\ y_1 = e_1 \ldots \textbf{ in}\ v) \longrightarrow_{Let} (\Gamma', y_1 \mapsto e_1 \ldots) : x \mapsto v$$

The inductive hypothesis (on the $\longrightarrow_A^*$) immediately gives the result for all value bindings in Γ'. All that we need to show is that x evaluates to v in Launchbury's semantics. That's easy:

$$\cfrac{\cfrac{(\Gamma, y_1 \mapsto e_1 \ldots) : v \Downarrow (\Gamma, y_1 \mapsto e_1 \ldots) : v}{\Gamma : \textbf{let } y_1 = e_1 \ldots \textbf{ in } v \Downarrow (\Gamma, y_1 \mapsto e_1 \ldots) : v}\ \textit{Let}}{\Gamma : x \Downarrow (\Gamma, y_1 \mapsto e_1 \ldots) : \hat{v}}\ \textit{Variable}$$

App:

$$[\Gamma] \longrightarrow_A^* (\Gamma', y \mapsto \lambda\, w.e) : x \mapsto y\, z \longrightarrow_{App} (\Gamma', y \mapsto \lambda\, w.e) : x \mapsto e[z/w]$$

Again the inductive hypothesis gives the result for all value bindings in Γ' and in particular:

$$\Gamma : y \Downarrow \Theta : \lambda\, w.e$$

If $e[z/w]$ is a value then we need to prove the result for x. It follows from the above by Launchbury's *Application rule*:

$$\frac{\Gamma : y \Downarrow \Theta : \lambda\, w.e \qquad \Theta : e[z/w] \Downarrow \Theta : e[z/w]}{\Gamma : y\, z \Downarrow \Theta : e[z/w]}\ \textit{Application}$$

Out/In:

$$[\Gamma] \longrightarrow_A^* (\Gamma', x \mapsto v) : y \mapsto e \longrightarrow_{O/I} (\Gamma', y \mapsto e) : x \mapsto v$$

Immediate for all value bindings by inductive hypothesis.
Var:

$$[\Gamma, x \mapsto y] \longrightarrow_A^* (\Gamma', y \mapsto v) : x \mapsto y \longrightarrow_{Var} (\Gamma', y \mapsto v) : x \mapsto \hat{v}$$

Again the inductive hypothesis gives the result for all value bindings except x. By the inductive hypothesis we have

$$(\Gamma, x \mapsto y) : y \Downarrow \Theta : v$$

Since this deduction succeeds it must be the case that we can also derive

$$\Gamma : y \Downarrow \Theta : v$$

Otherwise y would directly depend on itself before a value was returned, and *both* proofs would have failed. Now we can simply apply the *Variable* rule:

$$\frac{\Gamma : y \Downarrow \Theta : v}{(\Gamma, x \mapsto y) : x \Downarrow \Theta : \hat{v}}\ \textit{Variable}$$

$\square$

We informally discussed the relationship between parallelism and sequentiality in the context of infinite resources in the introduction to his section. A formal characterisation of this is:

Theorem 2. *If* $\mathbf{max} < \infty$ *then for any computation there is an equivalent computation in a system with a single resource.*

We first define the notion of a *sequential* computation:

Definition 1. *A computation is sequential when, for all judgments* $\Delta : \Gamma$ *which appear in the computation,* $|\Gamma| \leq 1$.

Sequential computations can take place with as little as a single resource available.

Whence:

Proof Idea: By induction on the length of the proof. Base case is immediate (as initial configuration is sequential). Induction step 'flattens' a lockstep into its components (with judicious us of *In* and *Out*). □

5 Discussion and Conclusions

We have defined a lockstep semantics of a functional language with explicit parallelism which models the effects of bounded resources during execution. We have shown that, in the case when there is a single resource, the system is the same as (to all intents and purposes) that of [Lau93]. One useful implication is that, at least in this restricted resource case, the system correctly behaves lazily, according to Launchbury's proof of consistency with Abramsky's semantics of [Abr90]. We have also shown that when there is more than a single resource each computation is serialisable (i.e., is equivalent to a computation on a machine with a single resource). In combination with the previous result we have that, even in the presence of parallelism, sharing is preserved, i.e., an expression is evaluated at most once.

However, the existing semantics is unsatisfactory because it does not identify what bindings should be moved from the dead heap to the live heap when resources are available. As a consequence, by selecting appropriate bindings each time it is possible to model a range of reduction strategies, including normal order and applicative order. On a machine with an unbounded number of processors it is even possible to model Knuth-Bendix reduction where every redex is reduced. An unfortunate consequence of this under-specification is that the semantics is highly speculative: it does not guarantee that a binding promoted to the live heap is needed by the program. By choosing inappropriate bindings an expression that would not be evaluated by a GpH program may be evaluated by the semantics.

Although there are many rules which could be added to the system which *reduce* the amount of speculative evaluation, we have been unable to find one that removes it completely (at least whilst preserving the resource consciousness of the semantics) without recording dependencies between active threads and the bindings needed or demanded by a thread (as occurs in the GpH run-time system!). We are actively investigating an augmented semantics that records these dependencies.

Such an augmented semantics, in removing (unplanned) speculative evaluation, will more accurately model GpH. With it, we will be in a better position to investigate the necessary notions of equivalence, such as simulation or bisimulation, which could serve for the characterisation of properties of the coordination primitives, such as those proffered in Section 1.1 and, ultimately, evaluation strategies themselves.

To do this we will need to consider standard performance measures such as *Work*, *Average Parallelism*, and *Runtime* ([EZL89]) of which, in the context of Figure 1, we make the following observations:

- **Work** (the total number of reductions performed by all threads in the computation): the right hand sides of equivalences (II) and (III) perform more work than the left hand sides because there is an additional reduction in each.
- **Average Parallelism** (the average number of active threads during the computation): the average parallelism of the right hand sides of equivalences (II) and (III) is greater than that of the left, again because extra reductions are involved.
- **Runtime** (the number of reductions for the program to complete, for conservative parallelism runtime is work divided by average parallelism): intuitively the runtime, R, of the left and right hand sides of equivalence (III) are the same because.

  $$\begin{aligned} &R[\![\ e_0\ ]\!] + max(R[\![\ e_1\ ]\!], R[\![\ e_2\ ]\!]) \\ &\qquad = max(R[\![\ e_0\ ]\!] + R[\![\ e_1\ ]\!], R[\![\ e_0\ ]\!] + R[\![\ e_2\ ]\!]) \end{aligned}$$

  The same is the case for equivalence (II), *mutatis mutandis.*

We note that, in all cases, (I) preserves its status as an equivalence.

5.1 On the *Product* rule

The existing product rule is very complex, because it fulfills several functions simultaneously:

1. Reduce each binding in the live heap simultaneously.
2. Move whnf bindings from the live heap into the dead heap (*deactivation*)
3. Move non-whnf bindings from the dead heap into the live heap (*activation*)
4. Constrain the size of the live heap to be less than the number of processors.

It is possible to use a simplified product rule that controls only parallel reduction (function 1), if we introduce a second, *scheduling* relation that controls activation, deactivation and resource constraint (properties 2, 3 and 4). Computation is then modelled by alternating reduction and scheduling relations.

Acknowledgments

This work was produce under the APSET project, funded by The Open University.

Many thanks to David Crowe, of the Applied Mathematics Department of the Open University, for his always perspicacious comments.

References

[ABB+97] L. Augustsson, B. Boutel, F.W. Burton, J.H. Fasel, A.D. Gordon, R.J.M. Hughes, P. Hudak, T. Johnsson, M.P. Jones, S.L. Peyton Jones, A. Reid, and P.L. Wadler. Report on the non-strict functional language, Haskell, version 1.4. Technical report, 1997.

[Abr90] Samson Abramsky. The Lazy Lambda Calculus. In D. Turner, editor, *Declarative Programming*. Addison-Wesley, 1990.

[BCH+93] G.E. Blelloch, S. Chatterjee, J.C. Hardwick, J. Spielstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *Proc. Fourth ACM Conf. on Principles & Practice of Parallel Programming (PPoPP)*, pages 102–111. San Diego, CA, May 1993.

[BLOMP97] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. The Eden coordination model for distributed memory systems. In *Proc. HIPS '97 — High-Level Parallel Programming Models and Supportive Environments*. IEEE Press, 1997.

[CGKL98] M.M.T. Chakravarty, Y. Guo, M. Köhler, and H.C.R. Lock. Goffin: Higher-order functions meet concurrent constraints. *Science of Computer Programming*, 30(1-2):157–199, 1998.

[EZL89] Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, March 1989.

[FMS+95] J. Feo, P. Miller, S. Skedziewlewski, S. Denton, and C. Soloman. Sisal 90. In *Proc. HPFC '95 — High Performance Functional Computing*, pages 35–47, April 1995. Denver, CO.

[Hal98] R. Halstead. MultiLisp: A language for concurrent symbolic computation. *ACM TOPLAS*, 7(4):501–538, 1998.

[Hen88] Matthew Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

[Hen90] Matthew Hennessy. *The Semantics of Programming Languages: An Elementary Introduction using Structural Operational Semantics.* Wiley, 1990.

[KG89] J.M. Kewley and K. Glynn. Evaluation annotation for Hope+. In *Proc. Glasgow Workshop on Functional Programming*, pages 329–337, August 1989.

[Lau93] John Launchbury. A natural semantics for lazy evaluation. In *Proc. of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, Charleson, South Carolina, January 1993. ACM SIGPLAN-SIGACT.

[MH95] R. Mirani and P. Hudak. First-class schedules and virtual maps. In *Proc. FPCA '95 — Functional Programming and Computer Architecture*, pages 78–85, June 1995.

[Mil89] A. J. C. Milner. *Communication and Concurrency.* Prentice-Hall International, 1989.

[MS95] G. Michaelson and N. Scaife. Prototyping a parallel vision system in Standard ML. *Journal of Functional Programming*, 5(3):345–382, 1995.

[NAH93] R.S. Nikhil, Arvind, and J. Hicks. pH language proposal. Technical report, DEC Cambridge Research Lab, 1993.

[NSvEP91] E.G.J.M.H. Nöcker, J.E.W. Smetsers, M.C.J.D. van Eekelen, and M.J. Plasmeijer. Concurrent Clean. In *Proc. PARLE '91 — Parallel Architectures and Reduction Languages Europe*, volume 505/506 of *LNCS*, pages 202–220. Springer Verlag, 1991.

[Plo81] G. D. Plotkin. Structured approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[THJ+96] P.W. Trinder, K. Hammond, J.S. Mattson Jr., A.S. Partridge, and S.L. Peyton Jones. GUM: A portable parallel implementation of Haskell. In *Proc. PLDI '96 — Programming Language Design and Implementation*, pages 78–88, May 1996.

[THLJ98] P.W. Trinder, K. Hammond, H.-W. Loidl, and S.L. Peyton Jones. Algorithm + strategy = parallelism. *Journal of Functional Programming*, 8(1), January 1998.