An Operational Semantics for Parallel Lazy Evaluation

Clem Baker-Finch
University of Canberra
clem@ise.canberra.edu.au

David J. King
Motorola UK Research Lab
David.King@motorola.com

Phil Trinder
Heriot-Watt University
trinder@cee.hw.ac.uk

10th March 2000

Abstract

We present an operational semantics for *parallel lazy evaluation* that accurately models the parallel behaviour of the non-strict parallel functional language GPH. Parallelism is modelled synchronously, that is, single reductions are carried out separately then combined before proceeding to the next set of reductions. Consequently the semantics has two levels, with transition rules for individual threads at one level and combining rules at the other. Each parallel thread is modelled by a binding labelled with an indication of its activity status. To the best of our knowledge this is the first semantics that models such thread states. A set of labelled bindings corresponds to a heap and is used to model sharing.

The semantics is set at a higher level of abstraction than an abstract machine and is therefore more manageable for proofs about programs rather than implementations. At the same time, it is sufficiently low level to allow us to reason about programs in terms of *parallelism* (i.e. the number of processors used) as well as *work* and *run-time* with different numbers of processors.

The framework used by the semantics is sufficiently flexible and general that it can easily be adapted to express other evaluation models such as sequential call-by-need, fully-speculative evaluation, non-deterministic choice and others.

1 Introduction

This paper provides an operational semantics that captures the *call-by-need* evaluation of a parallel extension to the λ-calculus. We use the term *call-by-need* synonymously with *lazy evaluation*, that is, normal order reduction to weak head normal form (whnf) with subexpression sharing. The extended λ-calculus language used in this paper, GPH-CORE, models Glasgow Parallel Haskell [39], an established parallel derivative of the non-strict purely functional language Haskell [30]. Parallelism is provided in both GPH and GPH-CORE in a *mostly-implicit* way, by using the annotation **par** to express parallel composition while leaving thread management to the compiler's run-time system. Note that **par** is only a hint of what to compute in parallel; it does not change the value returned.

The semantics presented is *explicit* in describing the way threads are managed and stored. Therefore, the semantics allows us to reason accurately about the behaviour of parallel functional programs in terms of *coordination* (i.e. how the computation is arranged in parallel) as well as *computation* (i.e. what value to compute).

A parallel call-by-need semantics is important for theoretical reasons but there are also several practical applications. For example, it provides a basis for applying *equational reasoning* to the parallel behaviour (or coordination) of parallel functional programs. For instance, it allows us to show properties such as: *program X uses more processors than program Y*; *program X uses more space than program Y*; and *program X runs faster than program Y*. Being able to reason about these properties can be useful for the programmer and compiler writer alike. For the programmer, they will be able to understand and improve the behaviour of their programs more easily. For the compiler writer, optimising transformations [13] and the verification of compiler models such as abstract machines and simulators can be justified.

The parallel call-by-need semantics presented here is a substantial development of earlier work [17] that gave a semantics of the same language but with the speculative evaluation of expressions in an undetermined order. The semantics presented here has the following key features:

- Name/expression bindings are used to model closures, which in turn are used to model threads. The bindings are labelled to model the GPH thread states: inactive, blocked, runnable and active.
- A heap of bindings is used to model space-usage, sharing and parallelism (i.e. processor usage can be quantified).
- The semantics is parameterised on the number of processors N, so we can readily model the behaviour of programs under differing resource assumptions.
- It is structured and therefore relatively simple to use for proving both computational and coordinational properties of programs. The simplicity arises because it models parallelism synchronously and the rules are at two levels: single thread transitions at one level and multi thread relations at the other level.
- It captures the thread behaviour and evaluation order of an existing parallel functional language implementation, GPH.

- The underlying idea of labelling heap bindings with their activity status is sufficiently flexible and powerful to describe a variety of other models of parallel lazy evaluation.

In Sections 2–11 we describe the framework of our operational semantic technique and develop a specification of the computation and coordination behaviour of GPH. In Section 12 we relate our semantics to a standard denotational semantics for sequential lazy evaluation and prove a determinacy result.

We examine several alternative models of parallel lazy evaluation to demonstrate the flexibility and generality of our approach in Section 13.

In Section 14 we define some standard metrics of parallelism in terms of our semantics. We conclude by discussing related work in Section 15 and our future plans in Section 16.

2 Modelling parallelism

The operational semantics we present is a two-level transition semantics. At the lower level, there are single-thread transitions for performing the ordinary evaluation of expressions (β-reduction and the like). All candidate single-thread steps are carried out simultaneously (conceptually on separate processors) and in lockstep. They are then combined into a parallel computation step using coordination relations defined at the upper level.

We follow Launchbury's seminal work [20] by using a heap to allow sharing and a restricted language to avoid the complications of creating closures in the semantics. The restricted language ensures that every closure is provided with a 'name' (a variable) by some corresponding let-binding. Our semantics is set at a lower level of abstraction than Launchbury's since we use a small-step computational semantics rather than a big-step natural semantics. This is the typical approach for modelling parallelism [18] since it allows us to more directly represent the coordination of multiple separate actions.

Nevertheless, our semantics is at a higher level than an abstract machine [5], thus avoiding the need for stacks, blocking queues and such like.

3 The language (GPH-CORE)

GPH-CORE is a simple subset of the language GPH. It consists of the *untyped λ-calculus* extended with numbers, recursive **let**s, a form of sequential composition **seq** and a form of parallel composition **par**. Since [20], it has become common practice to normalise the language to a restricted syntax. The normalised terms differ from their corresponding terms in GPH in two ways: all variables are distinct; and the second argument of application and the first argument of **par** must be variables. In fact, the normalised terms correspond to the internal core language that is used in the implementation of GPH. The process of normalisation is straightforward by introducing new let-bindings; for example, an algorithm is given in [17].

$$\begin{aligned} x, y, z &\in \text{Variable} \\ n &\in \text{Number} \\ e &\in \text{Expression} \\ e &::= n \mid x \mid e\ x \mid \lambda x.e \mid \mathbf{let}\ \{x_i = e_i\}_{i=1}^{n}\ \mathbf{in}\ e \\ &\quad \mid e_1\ \mathbf{seq}\ e_2 \mid x\ \mathbf{par}\ e \end{aligned}$$

Numbers are unnecessary but are included to make examples easier to follow. No extra rules are needed in the semantics to deal with numbers. In our semantics only closed terms will be considered, that is, programs contain no free variables. Constructors and case expressions are also a standard part of the GPH language but we will not include them here. They are unimportant to our central concern with parallelism.

3.1 Parallel and sequential composition

GPH-CORE and GPH express parallel coordination using the combinators **par** (for parallel composition) and **seq** (for sequential composition). These combinators are defined as follows:

$$e_1\ \mathbf{seq}\ e_2 = \begin{cases} \bot, & \text{if } e_1 = \bot \\ e_2, & \text{otherwise} \end{cases}$$
$$e_1\ \mathbf{par}\ e_2 = e_2$$

The operational behaviour of **seq** is to reduce e_1 to weak head normal form (whnf) before returning e_2, thus enforcing an evaluation order. The termination properties of a program can therefore be changed by using **seq**, in contrast to **par** which has no affect on termination properties. The operational behaviour of **par** is to proceed with the evaluation of e_2 and if a separate processor is available, to evaluate e_1 simultaneously. The idea of the **par** combinator is not new; its invention is attributed to John Hughes [36].

It is important to note that **par** will evaluate e_1 to whnf and no further. Hence the need for **seq** which is often used to force the evaluation of data structures to normal form. For example, if xs and ys are two lists then xs **par** ys will not evaluate the two lists in parallel with call-by-need evaluation because they are in whnf. Using **seq**, however, a function *seqList* can be defined that forces the evaluation of the structure of a list. Hence, to reduce the two lists in parallel and return them we can use ($seqList\ xs$ **par** $seqList\ ys$) **seq** (xs, ys). Trinder et al. [38] offers a much more detailed exposition of **par** and **seq**.

4 Heaps and labelled bindings

Following [20] we use a heap of bindings of expressions to variables, but for our purposes each binding also carries a label to indicate its state of activity. Thus, heaps are partial functions from variables to expression, thread state pairs:

$$\begin{aligned} H, K &\in \text{Heap} = \text{Variable} \hookrightarrow (\text{Expression}, \text{State}) \\ \alpha, \beta &\in \text{State} \\ \alpha &::= \mathit{Inactive} \mid \mathit{Runnable} \mid \mathit{Active} \mid \mathit{Blocked} \end{aligned}$$

We write individual bindings with the thread state appearing as an annotation on the binding arrow, thus:

$$x \overset{\alpha}{\mapsto} e$$

The binding state is usually be abbreviated to the first letter, I, R, A, or B. A binding is *Active* if it is currently being evaluated; it is *Blocked* if it is waiting for the evaluation of another binding before its evaluation can proceed; and it is *Runnable* if there are not enough resources currently available to evaluate it. All other bindings are *Inactive*, that is, they have not yet been initiated or they have finished

$$H : z \overset{A}{\mapsto} \textbf{let}\ \{x_i = e_i\}_{i=1}^n\ \textbf{in}\ e \ \longrightarrow\ (\{x_i \overset{I}{\mapsto} e_i\}_{i=1}^n, z \overset{A}{\mapsto} e) \qquad (let)$$

$$(H, x \overset{I}{\mapsto} v) : z \overset{A}{\mapsto} x \ \longrightarrow\ (z \overset{A}{\mapsto} \hat{v}) \qquad (var)$$

$$(H, x \overset{I}{\mapsto} e) : z \overset{A}{\mapsto} x \ \longrightarrow\ (x \overset{R}{\mapsto} e, z \overset{B}{\mapsto} x) \qquad (block_1)$$

$$(H, x \overset{RAB}{\mapsto} e) : z \overset{A}{\mapsto} x \ \longrightarrow\ (z \overset{B}{\mapsto} x) \qquad (block_2)$$

$$H : z \overset{A}{\mapsto} z \ \longrightarrow\ (z \overset{B}{\mapsto} z) \qquad (blackhole)$$

$$H : z \overset{A}{\mapsto} (\lambda y.e)\ x \ \longrightarrow\ (z \overset{A}{\mapsto} e[x/y]) \qquad (subst)$$

$$\frac{H : z \overset{A}{\mapsto} e \ \longrightarrow\ (K, z \overset{\alpha}{\mapsto} e')}{H : z \overset{A}{\mapsto} e\ x \ \longrightarrow\ (K, z \overset{\alpha}{\mapsto} e'\ x)} \qquad (app)$$

$$H : z \overset{A}{\mapsto} v\ \textbf{seq}\ e \ \longrightarrow\ (z \overset{A}{\mapsto} e) \qquad (seq\text{-}elim)$$

$$\frac{H : z \overset{A}{\mapsto} e_1 \ \longrightarrow\ (K, z \overset{\alpha}{\mapsto} e_1')}{H : z \overset{A}{\mapsto} e_1\ \textbf{seq}\ e_2 \ \longrightarrow\ (K, z \overset{\alpha}{\mapsto} e_1'\ \textbf{seq}\ e_2)} \qquad (seq)$$

$$(H, x \overset{RAB}{\mapsto} e_1) : z \overset{A}{\mapsto} x\ \textbf{par}\ e_2 \ \longrightarrow\ (z \overset{A}{\mapsto} e_2) \qquad (par\text{-}elim)$$

$$(H, x \overset{I}{\mapsto} e_1) : z \overset{A}{\mapsto} x\ \textbf{par}\ e_2 \ \longrightarrow\ (x \overset{R}{\mapsto} e_1, z \overset{A}{\mapsto} e_2) \qquad (par)$$

Figure 1: Single thread transition rules

being evaluated. A heap cannot contain multiple bindings for the same identifier.

The idea of labelling bindings with an indication of their state of activity is surprisingly powerful and general, allowing us to model a variety of parallel evaluation strategies as we demonstrate in Section 13. However, our main interest is in describing GPH and that is what we focus on for now.

Bindings correspond to heap closures and in our semantics labelled bindings correspond to threads. In the run-time system for GPH [39] only closures that are not inactive correspond to threads. GPH also uses other thread states such as *fetching*, which is the communication state. It is useful to be able to model communication, but it is not as important for GPH as it is for many other languages. Our semantics attributes no cost to communication, so an idealised architecture is modelled. Some profiling simulators for GPH, for example HBC-PP [35] and GRANSIM-Light [21], also model such an idealised machine and they have proven to give useful information.

The computational semantics is given as a relation on heaps $H \Longrightarrow H'$ which is in turn defined in terms of a notion of single thread transitions (Section 5) and a scheduling relation (Section 7).

The operational semantics describes a *reduction sequence* from an initial global configuration to a final global configuration:

$$(H, main \overset{A}{\mapsto} e) \Longrightarrow \ldots \Longrightarrow (H', main \overset{I}{\mapsto} v)$$

where *main* is always used as the program identifier. Hence, a program terminates as soon as *main* reaches a value v, irrespective of other bindings. Values v are whnf expressions in GPH-CORE, that is:

$$v ::= n \ \mid\ \lambda x.e$$

5 Single thread transitions

The transition function $\longrightarrow$ defined in Figure 1 describes the computational step taken by each active binding in the heap. The left hand side in each rule represents a heap with the particular active binding distinguished:

$$H : z \overset{A}{\mapsto} e$$

Several of the rules depend on other bindings in the heap (for example, the $block_i$ rules) so we also use the notation $(H, x \overset{\alpha}{\mapsto} e)$ to partition the heap, separating $x \overset{\alpha}{\mapsto} e$ from the other bindings in H. Multi-label bindings, such as $x \overset{RAB}{\mapsto} e$ in the $block_2$ rule mean that the state is one of B, A or R but not I.

The right hand sides of the rules in Figure 1 is a heap in the sense that it is a set of labelled bindings but it consists of *only those bindings that are changed or created by that computation step.* The changes for all active bindings are combined by the *parallel* rule (Section 6) to create a full heap to heap transition.

Let: The *let* rule populates the heap with new bindings. These bindings are inactive since under call-by-need they may not necessarily be evaluated.

Variables and blocking: In the *var* and $block_i$ rules the distinguished binding $z \overset{A}{\mapsto} x$ represents the situation of evaluating a closure (called z) which consists of a pointer to another closure (called x). If x has already been evaluated to whnf as in the *var* rule, then z simply receives that value.

The notation $\hat{v}$ is used to signify a renaming of all bound variables in v to fresh variable names. The *var* rule is the only place where names are duplicated. As Launchbury proves [20], this is sufficient to avoid all unwanted name clashes. An alternative suggested by Sestoft [37] is to rename bound variables at the time that let bindings are added to the heap. Sestoft's approach is preferred for abstract machines [5] but Launchbury's notation is more parsimonious and adequate for our present purposes.

If x is inactive and has not yet been evaluated, as in the *block*$_1$ rule, then z blocks at this point and x joins the pool of runnable bindings, possibly to be activated by the scheduling relation. If x is already active, runnable or blocked as in *block*$_2$ then z blocks but x is unaffected.

Black holes: We defer a detailed discussion of the *blackhole* rule to Section 11.

Application: Non-strict evaluation of $e_1\, e_2$ proceeds by reducing e_1 to an abstraction (*app* rule) and then substituting the argument x for the bound variable y (*subst*).

Seq: The *seq* rule proceeds to evaluate e_1 in the expression e_1 **seq** e_2 but makes no progress on e_2. When and if e_1 reaches whnf its value (but not the effect on the heap) is discarded by the *seq-elim* rule and evaluation proceeds to e_2.

Par: In GpH the only way to introduce parallelism is by using **par** and this is reflected in the semantics by the *par* rule. Notice that it does not create more parallelism immediately but instead suggests that a binding should be made active by putting it into a *Runnable* state which may be promoted to *Active* later. This promotion can happen in the scheduling phase (Section 7) but only if sufficient processing resources are available. This corresponds to GpH where **par** is thought of as a *hint* of what to parallelise, rather than a *command*. If the binding is already active, runnable or blocked there is no more to do (*par-elim*).

6 Multi-thread transitions

Active bindings are delegated for single thread steps by the following parallel computation rule, which bears some similarities with Milner's **Prod** rule [24]. This is the key point in the semantics where reductions are carried out in parallel. The rule also melds together all the new bindings, updating the heap accordingly (Figure 2).

$$\frac{H^A = \{\, x_i \overset{A}{\mapsto} e_i \,\}_{i=1}^n \qquad \{H : x_i \overset{A}{\mapsto} e_i \;\longrightarrow\; K_i\}_{i=1}^n}{H \overset{p}{\Longrightarrow} H[\bigcup_{i=1}^n K_i]} \qquad (\textit{parallel})$$

Figure 2: Combining multiple thread transitions

We write H^A to represent all the active bindings in H. That is:

$$H^A = \{x \overset{A}{\mapsto} e \in H\}$$

Hence in Figure 2 there are exactly n active bindings in H.

The notation $H[K]$ updates the heap H with all the new or changed bindings given by K. More precisely, it can be defined as follows:

$$H[K] = \{\, x \overset{\alpha}{\mapsto} e \in H \mid x \notin dom(K)\,\} \cup K$$

By taking the union of all the K_i in the *parallel* rule there appears to be the potential for conflicts to arise between bindings. What if $x \overset{B}{\mapsto} e \in K_1$ and $x \overset{A}{\mapsto} e' \in K_2$, for example? The following proposition demonstrate that for the single thread rules defined in Figure 1, no such conflict can arise.

Proposition 6.1
Given $H = H_1 : z_1 \overset{A}{\mapsto} e_1 = H_2 : z_2 \overset{A}{\mapsto} e_2$ where $z_1 \neq z_2$. If $H_1 : z_1 \overset{A}{\mapsto} e_1 \;\longrightarrow\; K_1$ and $H_2 : z_2 \overset{A}{\mapsto} e_2 \;\longrightarrow\; K_2$ then:

- if $x \overset{IAB}{\mapsto} e \in K_1$ then $x \notin dom(K_2)$
- if $x \overset{R}{\mapsto} e \in K_1$ then either $x \notin dom(K_2)$ or $x \overset{R}{\mapsto} e \in K_2$

Proof By induction on e_1 and e_2 and the syntactic restriction that all variables are distinct (Section 3). ∎

7 The scheduling relation

As well as the computational steps defined so far, we also need to describe the coordination aspects of the language. In particular we need to give the semantics of the scheduling phase of the evaluation. The scheduling actions for individual threads are defined in Figure 3 as follows:

- Any binding that is immediately blocked on a completing thread is made runnable (*unblock*);
- An active or runnable binding that is in whnf is made inactive because its evaluation is done (*deactivate*);
- As many runnable bindings as resources will allow are promoted to being active (*activate*).

$$(H, x \overset{RA}{\mapsto} v, z \overset{B}{\mapsto} e^x) \;\overset{u}{\longrightarrow}\; (H, x \overset{RA}{\mapsto} v, z \overset{R}{\mapsto} e^x) \qquad (\textit{unblock})$$

$$(H, x \overset{RA}{\mapsto} v) \;\overset{d}{\longrightarrow}\; (H, x \overset{I}{\mapsto} v) \qquad (\textit{deactivate})$$

$$\frac{|H^A| < \mathrm{N}}{(H, x \overset{R}{\mapsto} e) \;\overset{a}{\longrightarrow}\; (H, x \overset{A}{\mapsto} e)} \qquad (\textit{activate})$$

Figure 3: Single thread scheduling rules

The notation e^x represents an expression that is immediately blocked on x. In our language these can only take three forms:

$$e^x ::= x \mid x\ y \mid x\ \mathbf{seq}\ e'$$

In the *activate* rule in Figure 3, N is a parameter to the semantics, indicating the total number of processors. Hence we require that no more than N bindings are activated. Note that nowhere in these rules is it specified *which* bindings are activated. The activation phase presents a *non-deterministic* choice.

The rules in Figure 3 only affect single bindings. We need to unblock and deactivate *all* candidate threads and to activate as many as possible. To that end we make the definitions in Figure 4. In effect, the rules in Figure 4 define $\stackrel{\dagger}{\Longrightarrow}$ to be the *normal form* relations built upon $\stackrel{\dagger}{\longrightarrow}$.

$$H \stackrel{\dagger}{\Longrightarrow} H'$$

(where † represents u or d or a) if:

1. $H \stackrel{\dagger}{\longrightarrow}^* H'$ and
2. there is no H'' such that $H' \stackrel{\dagger}{\longrightarrow} H''$.

Figure 4: Component scheduling relations

Note that $\stackrel{a}{\Longrightarrow}$ is the only 'true' relation; $\stackrel{u}{\Longrightarrow}, \stackrel{d}{\Longrightarrow}, \stackrel{p}{\Longrightarrow}$ are all functions. In other words the only non-determinism that exists is in the choice of which threads from the runnable pool are activated.

Scheduling promotes runnable bindings into an active state if there are sufficient processors and demotes active evaluated expressions to an inactive state. To achieve maximum parallelism with respect to the number of processors it is necessary that all candidate threads are unblocked before deactivation and that deactivation takes place before activation to free up as many processors as possible. This sequence of actions is captured by the full scheduling relation defined in Figure 5.

$$\stackrel{s}{\Longrightarrow} \;=\; \stackrel{a}{\Longrightarrow} \circ \stackrel{d}{\Longrightarrow} \circ \stackrel{u}{\Longrightarrow} \qquad (\textit{schedule})$$

Figure 5: Complete scheduling relation

8 The computation relation

Finally the full computation relation of our semantics consists of a parallel transition ($\stackrel{p}{\Longrightarrow}$) followed by a scheduling of bindings ($\stackrel{s}{\Longrightarrow}$), as in Figure 6. A brief example is given in Section 9.

$$\Longrightarrow \;=\; \stackrel{s}{\Longrightarrow} \circ \stackrel{p}{\Longrightarrow} \qquad (\textit{compute})$$

Figure 6: Computation relation

Since the semantics is parameterised on the number of processors, we decorate the computation relation accordingly where necessary: ($\underset{\text{N}}{\Longrightarrow}$). In particular we use ($\underset{1}{\Longrightarrow}$) in Section 12 to indicate the single-processor case.

9 Example

To demonstrate the operational semantics in action we show in Figure 7 the reduction sequence with at least two processors for the following program:

$$\begin{aligned} \textit{main} = \;& \textbf{let} \;\; f = \lambda x.x \\ & \qquad g = \textbf{let}\; a = 3 \;\textbf{in}\; f\; a \\ & \textbf{in}\; g\; \textbf{par}\; (f\;\; g) \end{aligned}$$

Note that most steps in Figure 7 are expressed in terms of the full *compute* relation but in some interesting cases we separate the *parallel* and *schedule* steps. Underlines are used to emphasise the active bindings. As intended, f and g evaluate in parallel.

10 Keeping the main thread active

A consequence of the present definition of $\stackrel{a}{\Longrightarrow}$ is that the main thread may be left runnable but never progress. Suppose *main* blocks on some variable z by one of the blocking rules in Figure 1 and z is promoted to a runnable state. If the pool of runnable threads is larger than the number of available processors then there is no guarantee that z will be made active under the present *schedule* relation. It is possible that the main thread could thus be delayed or suspended indefinitely, if there is a constant supply of unneeded speculative threads being generated and scheduled in place of the main thread.

Yet, according to the designers of GpH, their implementation makes no assurance that a binding that *main* needs has priority over other bindings when deciding which ones to promote to an active state. While their early papers warn potential users that the run-time system offers little support for speculative evaluation [39], it nevertheless seems desirable for bindings required by the main thread to remain active whenever possible since that is the thread that delivers the final result. There may be no harm in suspending the main thread but there surely can be no advantage either.

We can model a solution to this 'defect' by modifying the *activate* relation $\stackrel{a}{\Longrightarrow}$ as in Figure 8 to require that any runnable thread on which *main* is blocked, in a transitive sense made precise by the function *pre*, is activated in preference to any other runnable thread. We can be sure that there will always be a free processor in this circumstance because the blocking action has made one available.

$$H \stackrel{a'}{\Longrightarrow} H' \quad \text{if:}$$

1. $H \stackrel{a}{\longrightarrow}^* H'$;
2. there is no H'' such that $H' \stackrel{a}{\longrightarrow} H''$ and
3. $\textit{pre}(\textit{main}, H')$ is active in H'.

$$\textit{pre}(x, K) = \begin{cases} x, & \text{if } x \stackrel{RA}{\mapsto} e \in K \\ \textit{pre}(y, K), & \text{if } x \stackrel{B}{\mapsto} e^y \in K \end{cases}$$

Figure 8: Stronger activation relation

$$
\begin{aligned}
&\{\, main \overset{A}{\mapsto} \textbf{let } f = \lambda x.x, g = (\textbf{let } a = 3 \textbf{ in } f\ a) \textbf{ in } g \textbf{ par } (f\ g) \,\} \\
&\Longrightarrow \{\, f \overset{I}{\mapsto} \lambda x.x, g \overset{I}{\mapsto} \textbf{let } a = 3 \textbf{ in } f\ a, \underline{main \overset{A}{\mapsto} g \textbf{ par } (f\ g)} \,\} && (let) \\
&\overset{p}{\Longrightarrow} \{\, f \overset{I}{\mapsto} \lambda x.x, g \overset{R}{\mapsto} \textbf{let } a = 3 \textbf{ in } f\ a, \underline{main \overset{A}{\mapsto} f\ g} \,\} && (par) \\
&\overset{s}{\Longrightarrow} \{\, f \overset{I}{\mapsto} \lambda x.x, \underline{g \overset{A}{\mapsto} \textbf{let } a = 3 \textbf{ in } f\ a}, \underline{main \overset{A}{\mapsto} f\ g} \,\} \\
&\Longrightarrow \{\, f \overset{I}{\mapsto} \lambda x.x, a \overset{I}{\mapsto} 3, \underline{g \overset{A}{\mapsto} f\ a}, \underline{main \overset{A}{\mapsto} (\lambda x.x)\ g} \,\} && (let,\ app_{(var)}) \\
&\Longrightarrow \{\, f \overset{I}{\mapsto} \lambda x.x, a \overset{I}{\mapsto} 3, \underline{g \overset{A}{\mapsto} (\lambda x.x)\ a}, \underline{main \overset{A}{\mapsto} g} \,\} && (app_{(var)},\ subst) \\
&\Longrightarrow \{\, f \overset{I}{\mapsto} \lambda x.x, a \overset{I}{\mapsto} 3, \underline{g \overset{A}{\mapsto} a}, main \overset{B}{\mapsto} g \,\} && (subst,\ block_2) \\
&\overset{p}{\Longrightarrow} \{\, f \overset{I}{\mapsto} \lambda x.x, a \overset{I}{\mapsto} 3, \underline{g \overset{A}{\mapsto} 3}, main \overset{I}{\mapsto} g \,\} && (var) \\
&\overset{u}{\Longrightarrow} \{\, f \overset{I}{\mapsto} \lambda x.x, a \overset{I}{\mapsto} 3, \underline{g \overset{A}{\mapsto} 3}, main \overset{R}{\mapsto} g \,\} \\
&\overset{d}{\Longrightarrow} \{\, f \overset{I}{\mapsto} \lambda x.x, a \overset{I}{\mapsto} 3, g \overset{I}{\mapsto} 3, main \overset{R}{\mapsto} g \,\} \\
&\overset{a}{\Longrightarrow} \{\, f \overset{I}{\mapsto} \lambda x.x, a \overset{I}{\mapsto} 3, g \overset{I}{\mapsto} 3, \underline{main \overset{A}{\mapsto} g} \,\} \\
&\Longrightarrow \{\, f \overset{I}{\mapsto} \lambda x.x, a \overset{I}{\mapsto} 3, g \overset{I}{\mapsto} 3, main \overset{I}{\mapsto} 3 \,\} && (var)
\end{aligned}
$$

Figure 7: Example reduction

11 Black holes

Black holes are such detectable infinite loops as:

$$
\begin{aligned}
&\textbf{let } x = x \textbf{ in } x \\
&\textbf{let } x = y, y = z, z = x \textbf{ in } x
\end{aligned}
$$

How should our semantics treat these expressions? Taking the simplest case of direct self-reference, we have:

$$
\begin{aligned}
(H, z \overset{A}{\mapsto} \textbf{let } x = x \textbf{ in } x) &\Longrightarrow (H', x \overset{I}{\mapsto} x, z \overset{A}{\mapsto} x) \\
&\Longrightarrow (H', x \overset{A}{\mapsto} x, z \overset{B}{\mapsto} x)
\end{aligned}
$$

Without the *blackhole* rule of Figure 1 we can go no further because there is no single thread rule that applies to an active thread like $x \overset{A}{\mapsto} x$ and the *parallel* rule fails since it requires all active bindings to progress. In general that is not what we want because other threads may still be doing useful work towards the final result. The *blackhole* rule deals with this situation by converting $x \overset{A}{\mapsto} x$ to blocked $x \overset{B}{\mapsto} x$, thereby releasing its processor for other useful evaluation. This behaviour corresponds to the implementation of GPH.

Cycles such as $\textbf{let } x = y, y = x \textbf{ in } x$ that lead to black holes also become blocked in our semantics, but without the need for a special rule:

$$
\begin{aligned}
&(H, z \overset{A}{\mapsto} \textbf{let } x = y, y = x \textbf{ in } x) \\
\Longrightarrow&(H', x \overset{I}{\mapsto} y, y \overset{I}{\mapsto} x, z \overset{A}{\mapsto} x) \\
\Longrightarrow&(H', x \overset{A}{\mapsto} x, y \overset{I}{\mapsto} x, z \overset{B}{\mapsto} x) \\
\Longrightarrow&(H', x \overset{B}{\mapsto} x, y \overset{A}{\mapsto} x, z \overset{B}{\mapsto} x) \\
\overset{p}{\Longrightarrow}&(H', x \overset{B}{\mapsto} x, y \overset{B}{\mapsto} x, z \overset{B}{\mapsto} x)
\end{aligned}
$$

If the *main* thread depends on a black hole and we require it to remain active as described in Section 10 then we would expect the computation to halt. In such a case $pre(main, H)$ is undefined so the *activate'* rule fails. In other words, $(H \overset{a}{\Longrightarrow})$ is empty and hence so is $(H \Longrightarrow)$.

12 Properties of the semantics

Abramsky's and Ong's [1, 29] denotational semantics of lazy evaluation models functions by a lifted function space, thus distinguishing between a term Ω (a non-terminating computation) and $\lambda x.\Omega$ to reflect the fact that reduction is to weak head normal form rather than head normal form.

Launchbury [20] shows a number of results relating his natural semantics of lazy evaluation to this denotational semantics. We borrow much of his notation and several of our proofs are inspired by his. In earlier work [6] we showed that our semantics agrees with Launchbury's but here we take a more direct approach, working with the denotational model.

There are three main properties that we expect of our semantics: *correctness*: the computation relation preserves the meanings of terms; *adequacy*: evaluations terminate if and only if their denotation is not $\perp$; *determinacy*: the same result is always obtained, irrespective of the number of processors and irrespective of which runnable threads are chosen for activation during the computation. The determinacy result will only hold if the scheduling phase uses rule (*activate'*) which guarantees that the binding on which the main thread is blocked remains active, as discussed in Section 10.

The denotational semantics of our language is given in Figure 9. The *Val* domain is assumed to contain a lifted version of its own function space. The lifting injection is *lift* and the corresponding projection is *drop*.

The semantic function:

$$[\![\ldots]\!] : \mathit{Exp} \to \mathit{Env} \to \mathit{Val}$$

naturally extends to operate on heaps, the operational counterpart of environments:

$$\{\!\{ \ldots \}\!\} : \mathit{Heap} \to \mathit{Env} \to \mathit{Env}$$

$$\rho \in Env = Var \to Val$$

$$\begin{aligned}
[\![\lambda x.e]\!]_\rho &= \mathit{lift}\ \lambda\epsilon.[\![e]\!]_{\rho[x\mapsto\epsilon]}\\
[\![e\ x]\!]_\rho &= \mathit{drop}([\![e]\!]_\rho)([\![x]\!]_\rho)\\
[\![x]\!]_\rho &= \rho(x)\\
[\![\mathbf{let}\ \{x_i = e_i\}_{i=1}^n\ \mathbf{in}\ e]\!]_\rho &= [\![e]\!]_{\{\!\{x_1\mapsto e_1\ldots x_n\mapsto e_n\}\!\}\rho}\\
[\![e_1\ \mathbf{seq}\ e_2]\!]_\rho &= \begin{cases}\bot & \text{if } [\![e_1]\!]_\rho = \bot\\ [\![e_2]\!]_\rho & \text{otherwise}\end{cases}\\
[\![x\ \mathbf{par}\ e]\!]_\rho &= [\![e]\!]_\rho
\end{aligned}$$

Figure 9: Denotational semantics

The recursive nature of heaps is reflected by a recursively defined environment:

$$\{\!\{x_1 \mapsto e_1 \ldots x_n \mapsto e_n\}\!\}\rho = \mu\rho'.\rho[x_1 \mapsto [\![e_1]\!]_{\rho'} \ldots x_n \mapsto [\![e_n]\!]_{\rho'}]$$

We also require an ordering on environments: if $\rho \leq \rho'$ then ρ' may bind more variables than ρ but they are otherwise equal. That is:

$$\forall x\ .\ \rho(x) \neq \bot \Rightarrow \rho(x) = \rho'(x)$$

The arid environment ρ_0 takes all variables to $\bot$.

Correctness. Our computational relation $H \Longrightarrow H'$ can be considered correct if the denotations of all the bindings in H are unchanged in H'. The $\leq$ ordering on environments neatly captures this notion.

Proposition 12.1
If $H \Longrightarrow H'$ then for all ρ, $\{\!\{H\}\!\}\rho \leq \{\!\{H'\}\!\}\rho$

Proof By induction on the size of H and on the structure of e. ■

Adequacy. We wish to characterise the termination properties of our semantics and Propositions 12.2 and 12.3 show an agreement with the denotational definition. The proofs are modelled on the corresponding ones in [20].

Proposition 12.2
If $(H, z \overset{A}{\mapsto} e) \Longrightarrow^* (H', z \overset{I}{\mapsto} v)$ then $[\![e]\!]_{\{\!\{H\}\!\}\rho} \neq \bot$.

Proof For all values v, $[\![v]\!]_{\{\!\{H'\}\!\}\rho} \neq \bot$ so by Proposition 12.1 $[\![e]\!]_{\{\!\{H\}\!\}\rho} \neq \bot$. ■

Proposition 12.3
If $[\![e]\!]_{\{\!\{H\}\!\}\rho} \neq \bot$ then there exists H', z, v such that $(H, z \overset{A}{\mapsto} e) \Longrightarrow^* (H', z \overset{I}{\mapsto} v)$.

Our proof of Proposition 12.3 is outlined in Appendix A. It is closely based on the corresponding proof in [20], working with a variant of the denotational semantics which is explicit about finite approximations.

Determinacy. We now turn to the question of obtaining the same result irrespective of the number of processors and irrespective of which runnable threads are chosen for activation during the computation. Clearly, since the results above hold for any number of processors it follows that *if* an evaluation with N processors gives *main* a value then, depending on which threads are activated, an evaluation with M processors *can* give the same result in the sense of Proposition 12.1.

Without the side condition discussed in Section 10 that is the best that can be expected — otherwise in general it is possible for *main* to remain blocked indefinitely. With the side condition, we want to demonstrate that if *any* evaluation gives an answer for *main* then they all do, irrespective of the number of processors. For the 1-processor case, it is clear that the definition of $\overset{a}{\Longrightarrow}$ ensures that there is always exactly one active binding and that the blocked bindings form a chain from *main* to that active binding.

The following proposition demonstrates that all the closures activated in the one processor case will also be activated in the multi-processor case. Recall that $(\underset{N}{\Longrightarrow})$ is the computation relation assuming a maximum of N processors.

Proposition 12.4
Given some number of processors $N \geq 1$, suppose:

$$(H, \mathit{main} \overset{A}{\mapsto} e) \underset{1}{\Longrightarrow} H_1 \underset{1}{\Longrightarrow} H_2 \ldots$$

and:

$$(H, \mathit{main} \overset{A}{\mapsto} e) \underset{N}{\Longrightarrow} K_1 \underset{N}{\Longrightarrow} K_2 \ldots$$

If x is active in some H_i then there is a j such that x is active in K_j.

Proof Suppose z_k is active in some H_i. By *activate'* there is a chain $\mathit{main} \overset{B}{\mapsto} e^{z_1}, z_1 \overset{B}{\mapsto} e^{z_2}, z_2 \overset{B}{\mapsto} e^{z_3}, \ldots z_k \overset{A}{\mapsto} e$ in H_i.

By induction on the length k of this chain we can show that there must be some K_j where z_k is active in K_j. ■

Finally we can bring all these results to bear to prove that evaluation is deterministic in the sense that we get the same answer every time, for any number of processors, assuming the *activate'* rule.

Corollary 12.5
Suppose $(H, \mathit{main} \overset{A}{\mapsto} e) \underset{1}{\Longrightarrow}^* (H', \mathit{main} \overset{I}{\mapsto} v)$.
For any number of processors N, if:

$$(H, \mathit{main} \overset{A}{\mapsto} e) \underset{N}{\Longrightarrow} K_1 \underset{N}{\Longrightarrow} K_2 \ldots$$

then:

1. there is some $i \geq 1$ such that $K_i = (K_i', \mathit{main} \overset{I}{\mapsto} v')$;
2. $[\![v']\!]_{\{\!\{K_i'\}\!\}\rho_0} = [\![v]\!]_{\{\!\{H'\}\!\}\rho_0}$

Proof

1. If there is no such K_i then *main* must remain active or blocked forever. In either case there must be some binding $z \overset{A}{\mapsto} e$ that remains active and does not terminate. In that case the denotation of e in the context of the corresponding heap must be $\bot$ by Proposition 12.3. But by Proposition 12.4 at some stage in the

1-processor evaluation z will be active and $main$ will be (transitively) blocked on z. By Proposition 12.2 e will not reach a whnf so $main$ will remain blocked. (Unless $main = z$ in which case the result follows immediately.)

2. $\{\!\{H, main \overset{A}{\mapsto} e\}\!\}\rho_0 \leq \{\!\{H', main \overset{I}{\mapsto} v\}\!\}\rho_0$ by Proposition 12.1, so in particular $[\![v]\!]_{\{\!\{H'\}\!\}\rho_0} = [\![e]\!]_{\{\!\{H\}\!\}\rho_0}$.

   Similarly, $[\![v']\!]_{\{\!\{K'_i\}\!\}\rho_0} = [\![e]\!]_{\{\!\{H\}\!\}\rho_0}$.

■

13 Other approaches to parallel call-by-need

So far we have concentrated on a detailed treatment of the semantics of a particular parallel language, GpH, which is the focus of our broader current research program. However, our central framework of a heap of bindings labelled with an indication of their activity status is much more general than that, allowing us to describe various other models of parallel evaluation. There is insufficient space for a full treatment here but we hope to give enough information that the reader could fill in the details.

13.1 Sequential evaluation

Our first example is not parallel at all but simply sequential lazy evaluation. This can be achieved by just restricting the computation relation to the case of a single processor ($\underset{1}{\Longrightarrow}$) but a number of simplifications become possible. First there is no need for the notion of runnable bindings because there is always exactly one active binding: $pre(main, H)$. In that case we can eliminate the scheduling relation altogether by having the $block_1$ rule directly activate the inactive binding:

$$(H, x \overset{I}{\mapsto} e) : z \overset{A}{\mapsto} x \longrightarrow (z \overset{B}{\mapsto} x, x \overset{A}{\mapsto} e) \qquad (block'_1)$$

Of course there is also no need for the *parallel* rule. Full details are given in [6]. While there are other small-step semantics for lazy evaluation (e.g. [26]) this one may be of interest for the fact that it directly represents the passing of control between closures as they are evaluated.

13.2 Fully speculative evaluation

Fully speculative evaluation is a completely implicit approach where every application $e_1\ e_2$ introduces parallelism by proceeding to evaluate both e_1 and e_2 together [15]. This can easily be expressed in our framework by a modification to the *app* rule:

$$\frac{(H, x \overset{I}{\mapsto} v) : z \overset{A}{\mapsto} e \longrightarrow (K, z \overset{\alpha}{\mapsto} e')}{(H, x \overset{I}{\mapsto} v) : z \overset{A}{\mapsto} e\ x \longrightarrow (K, z \overset{\alpha}{\mapsto} e'\ x)} \qquad (app_1)$$

$$(H, x \overset{I}{\mapsto} e_2) : z \overset{A}{\mapsto} e_1\ x \longrightarrow (x \overset{R}{\mapsto} e_2, z \overset{A}{\mapsto} e_1\ x) \qquad (app_2)$$

$$\frac{(H, x \overset{RAB}{\mapsto} e_2) : z \overset{A}{\mapsto} e_1 \longrightarrow (K, z \overset{\alpha}{\mapsto} e'_1)}{(H, x \overset{RAB}{\mapsto} e_2) : z \overset{A}{\mapsto} e_1\ x \longrightarrow (K, z \overset{\alpha}{\mapsto} e'_1\ x)} \qquad (app_3)$$

Thus if x is an inactive, unevaluated closure then it is made runnable (app_2). Otherwise the evaluation proceeds in a fashion entirely analogous to Figure 1.

13.3 Non-deterministic choice

So far all our languages have been deterministic in the sense that the final result will always be the same (Proposition 12.1) but our framework can also describe non-deterministic choice operators such as McCarthy's **amb** [23, 26]. In general terms, to evaluate e_1 **amb** e_2, evaluate e_1 and e_2 in parallel and accept the first to terminate as the result. The only real complication for our semantics is that if e_1 terminates we wish to *deactivate* e_2 *and all the threads that it has spawned* (and vice versa if e_2 terminates first). The approach we take is to modify the unblocking component of the scheduling relation. First we give the single-thread transition for **amb**:

$$(H, x \overset{\alpha}{\mapsto} e_1, y \overset{\beta}{\mapsto} e_2) : z \overset{A}{\mapsto} x\ \mathbf{amb}\ y \longrightarrow$$
$$(x \overset{\alpha'}{\mapsto} e_1, y \overset{\beta'}{\mapsto} e_2, z \overset{B}{\mapsto} x\ \mathbf{amb}\ y) \qquad (amb)$$

where

$$\alpha' = \begin{cases} R & \text{if } \alpha = I \\ \alpha & \text{otherwise} \end{cases}$$

β' is defined similarly. Effectively, z blocks and x and y become runnable unless they are already runnable, active or blocked.

Now unblocking an **amb** expression is special because we want to kill the other arm:

$$(H, x \overset{RA}{\mapsto} v, z \overset{B}{\mapsto} x\ \mathbf{amb}\ y) \overset{u}{\longrightarrow}$$
$$((kill\ \ y\ H), x \overset{RA}{\mapsto} v, z \overset{B}{\mapsto} x)) \qquad (unblock_{amb})$$

There is a symmetrical rule for $y \overset{RA}{\mapsto} v$. The *kill* y function searches out all threads spawned by y and makes them inactive. We do not give a definition here but simply note that it has similarities to *pre* in that it follows chains of blocked bindings.

There is a slight complication however: it would be incorrect to make such a binding inactive in the case where another, possibly mandatory, thread is also blocked on the same binding. There is already sufficient structure and information in the heap to correctly define *kill* but it is much easier if we also record at least the number of threads blocked on each binding (cf. [5] where we maintain explicit blocking queues). Then *kill* can check that there are no others blocking on the binding before making it inactive.

13.4 Controlled speculative evaluation

As mentioned in Section 10, GpH does not provide much support for speculative evaluation. Here we sketch an extension to the language to put some control of speculative evaluation in the hands of the programmer.

The idea is to have a syntactic variant of **par**, say **spar** with the same denotational semantics as **par** but the runtime system (i.e. our *schedule* relation) is to give priority to non-speculative bindings (created by **par**) over speculative bindings (created by **spar**). We can achieve this by having another label on bindings: *Speculative* or *Non-speculative* (contracted to S and N) as well as the activity labels. There is not space for all the details here but the S, N labels are introduced by **spar** and **par** respectively. For example:

$$(H, x \overset{I}{\mapsto} e_1) : z \underset{\sigma}{\overset{A}{\mapsto}} x\ \mathbf{spar}\ e_2 \longrightarrow (z \underset{\sigma}{\overset{A}{\mapsto}} e_2, x \underset{S}{\overset{R}{\mapsto}} e_1) \quad (spar)$$

where σ is either S or N. The speculation labels are passed on when blocking occurs but with N taking priority over S in clashing cases.

During activation, if there are insufficient processors to activate all runnable bindings, enough speculative active bindings should be demoted to runnable to allow as many as possible non-speculative, runnable threads to be promoted to active. The details are not particularly difficult.

A natural generalisation of this model is to associate a *priority* with each thread created by **par** and the scheduler selects which threads to activate, based on this value [21]. The framework outline here can be modified to handle this approach by replacing the S and N labels with a numeric priority indicator.

14 Metrics of parallelism

One of the chief motivating factors for developing our semantics was to provide a means for formally comparing programs in terms of time and parallelism. It turns out to be straightforward to define common metrics for parallelism: work done, efficiency and speedup (see [12] for the standard definitions) in terms of a reduction sequence. How realistic is the operational semantics compared with a real compiler? Obviously our semantics is far removed from a real compiler, that will have a highly optimised reduction engine and we should not expect reductions in the semantics to compare precisely with compiled code. We do believe, however, that our level of abstraction is such that we can draw conclusions about parallelism and make reasonable comparisons about run-time. In fact, profiling simulators such as HBC-PP [35] are based on counting similar reductions to those in our semantics and have proven to give useful performance measurements.

Definition 14.1 (Work and run-time)
Given a reduction sequence $H_0 \Longrightarrow H_1 \Longrightarrow \cdots \Longrightarrow H_{t_{\text{N}}}$ for a terminating program with N processors, the total work done with respect to N processors is:

$$W(\text{N}) = \sum_{i=0}^{t_{\text{N}}} \left| H_i^A \right|$$

Run-time is simply given by:

$$R(\text{N}) = t_{\text{N}}$$

Definition 14.2 (Average parallelism)
Given an unbounded number of processors, the average parallelism is the total number of single thread transitions:

$$\text{Average parallelism} = \frac{W(\infty)}{R(\infty)}$$

In practice a definition parameterised on the number of processors can be more useful:

$$P(\text{N}) = \frac{W(\text{N})}{t_{\text{N}}}$$

Definition 14.3 (Maximum parallelism)
Given an unbounded number of processors, the maximum parallelism is the maximum number of active processors during evaluation:

$$\text{Maximum parallelism} = \max\{ \left| H_i^A \right| \}_{i=0}^{t_\infty}$$

Or again we can take account of the number of processors:

$$M(\text{N}) = \max\{ \left| H_i^A \right| \}_{i=0}^{t_{\text{N}}}$$

For the example in Section 9 with two or more processors the work done is 10 and the run-time is 7 giving an average parallelism of $1\frac{3}{7}$. Using the same example with 1 processor the run-time increases to 10, so using extra processors can improve the program's run-time.

15 Related work

There have been a variety of semantic descriptions for call-by-need and call-by-name, for example [31, 29, 19, 1, 34, 32, 3, 22, 20, 26]. This contrasts with a relative scarcity of semantic descriptions of parallel models. Parallel semantics of the λ-calculus have been explored with denotational semantics; for example Roe [34] defines a denotational semantics such that operational costs of parallel computations can be given. Greiner and Blelloch [15, 14] develop Roe's ideas further in a semantics that describes a fully-speculative reduction model for the λ-calculus. All of these semantics are useful for reasoning about different reduction models of the λ-calculus but none model a real parallel functional language. A unique feature of our semantics is that it models the behaviour of threads during a parallel reduction sequence of a real parallel functional language, namely GpH.

Other parallel or concurrent functional languages that have a defined semantics include NESL [9], Eden [11], pH [2, 4], Concurrent ML [33, 7] and Scheme [28]. All of these languages are based on the λ-calculus but include a different set of language features to GpH. Eden and Concurrent ML have explicit constructs for concurrency. NESL provides parallelism *implicitly* by implementing several language primitives using nested data-parallelism. pH has synchronisation, barriers and side-effects, all of which require special care in an operational semantics. Only Eden is call-by-need. The form of parallelism in GpH is mostly implicit, which makes it easier for the user to write parallel programs than more explicit languages. GpH is more expressive than purely implicit languages like NESL, because in GpH control parallelism as well as data parallelism can be expressed. There is no consensus on the best way of introducing parallelism to functional programming languages but the model described in this paper is used by a popular Haskell compiler (GHC) and, as we have demonstrated, our framework extends to other parallel evaluation strategies.

16 Further work

There are numerous opportunites to develop this work, beyond the consideration of other models of parallelism that we surveyed in Section 13.

16.1 Extending the semantics

Adding case, constructors and primitives is simple and left out of this paper because they are not relevant to the discussion of parallelism. There are three main extensions to the semantics that we are currently considering: modeling space, communication and asynchronous reduction.

Space. Currently we have a simple model that essentially uses a monolithic heap for all the storage space. This does not correlate very well with typical parallel compilers that distribute data across processors. For instance, GPH uses a separate heap on each processor and communicates threads between heaps. Others have worked on modelling space efficiently including Blumofe and Leiserson [10] and Blelloch et al. [8].

Communication. Another desirable extension is to model communication, which again requires a more detailed treatment of heap space, since the essence of communication is to transfer bindings between heaps on different processors. The semantics currently allows us to reason about four thread states (active, inactive, runnable and blocked) but we would also like to reason about other thread states such as *fetching* which is the communication state. This is important for modelling the behaviour of a program on a real parallel architecture whose processor topology may have an impact on a program's run-time. We have considered constructing models where communication latencies are modelled by using 'clocks' (counters) on compute steps.

Asynchronous reduction. We would like to extend our semantics to model the asynchronous reduction that is carried out by the GPH compiler. The clocking mechanism mentioned above for modelling communication seems to provide a simple mechanism for modelling asynchronous behaviour without adversely affecting the simplicity of the present semantics.

16.2 Equational reasoning for parallel coordination

Functional languages are promoted as for being good for equational reasoning, so that properties of programs are easily demonstrated by using familiar techniques. There is an abundance of work on equational reasoning about computation but very little on equational reasoning for parallel coordination and computation, perhaps because it is much more difficult to reason about parallel programs. In Section 14 we described how programs can be compared in terms of time and parallelism. This can be used to develop a family of *equivalences* and *cost orderings* that have a known behaviour, shown formally using the semantics. For example, the following two expressions are equivalent in terms of resource usage:

$$x \ \mathbf{seq}\ (y \ \mathbf{par}\ z) \qquad (x \ \mathbf{seq}\ y) \ \mathbf{par}\ (x \ \mathbf{seq}\ z).$$

While we have used the semantics to show the correctness of simple terms, much work needs to be done to develop a family of axioms for **par** and **seq** as well as higher-order combinators. Recent work by Sands, Moran and Gustavsson [25, 16] on operational techniques for call-by-need seems to hold some promise.

16.3 Abstract machines

Work is in progress, using this semantics for developing an abstract machine for lazy parallel graph reduction [5] just as Sestoft [37] did for Launchbury's sequential call-by-need semantics and Moreau did for Scheme with futures [27]. We have also written interpreters for our semantics [6] that allow us to analyse the behaviour of an expression automatically. The abstract machine allows us to study more closely the behaviour of a real compiler and low-level features such as unblocking techniques [15] and garbage-collection [40]. It should also be possible to use the semantics at this level to formally justify the behaviour of parallel simulators such as HBC-PP [35] and GRANSIM [21].

17 Conclusion

We have developed an operational semantics for parallel lazy evaluation that models the language GPH. The semantics uses the mechanism of a heap of bindings labelled with an indication of their activity status, to model sharing and parallel thread behaviour. The result is simpler than many other attempts at a parallel semantics. We were also able to show the correctness of our semantics with respect to standard sequential call-by-need semantics.

The techniques developed here are sufficiently powerful and flexible to describe a wide variety of approaches to parallel evaluation, as we demonstrated with fully-speculative evaluation, non-deterministic choice and programmer-controlled speculation. The flexibility carries over to a class of parallel lazy abstract machines we have derived from our semantic framework.

Developing the semantics has been successful in uncovering subtleties in the real implementation. It is far easier to reason about behaviours such as prioritising main thread bindings, black holes, unblocking techniques and so on, at this level of abstraction rather than in a real compiler.

Acknowledgments

We would like to thank Jon Hall for his contributions in the early stages of this project. We have also benefited from discussions with Dave Sands.

References

[1] Samson Abramsky. The lazy lambda calculus. In *Research Topics in Functional Programming*, pages 65–117. Addison Wesley, 1990.

[2] Shail Aditya, Arvind, Lennart Augustsson, Jan-Willem Maessen, and Rishiyur S. Nikhil. Semantics of pH: A parallel dialect of Haskell. In *Proceedings of the Haskell Workshop*, pages 35–49, 1995.

[3] Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3):265–301, 1997.

[4] Arvind, Jan-Willem Maessen, Rishiyur S. Nikhil, and Joe E. Stoy. λ_s an implicitly parallel λ-calculus with letrec, synchronization and side-effects. memo 393, MIT Laboratory for Computer Science, 1996.

[5] Clem Baker-Finch. An abstract machine for parallel lazy evaluation. In *Trends in Functional Programming*, chapter 17. Intellect, 2000. To appear.

[6] Clem Baker-Finch, David J. King, Jon G. Hall, and Phil Trinder. An operational semantics for parallel call-by-need. Technical Report No. 99/1, Department of Computing, The Open University, Milton Keynes, 1999.

[7] Dave Berry, Robin Milner, and David N. Turner. A semantics for ML concurrency primitives. In *The 19th Symposium on Principles of Programming Languages (POPL'92)*, pages 119–129, 1992.

[8] Guy E. Blelloch, Phillip B. Gibbons, Yossi Matias, and Girija J. Narlikar. Space-efficient scheduling of parallelism with synchronization variables. In *Proceedings of 9th ACM Symposium on Parallel Algorithms and Architectures (SPAA'97)*, pages 12–23, 1997.

[9] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *International Conference on Functional Programming (ICFP'96)*, pages 213–225, 1996.

[10] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal of Computing*, 27(1):202–229, 1998.

[11] Silvia Breitinger, Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña. *Eden: Language Definition and Operational Semantics.* Fachbereich Mathematik und Informatik, Philipps Universität Marburg, 1998. Technical Report 96-10.

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

[13] Cormac Flanagan and Matthias Felleisen. The semantics of future and its use in program optimization. In *The 22nd Symposium on Principles of Programming Languages (POPL'95)*, pages 209–220, 1995.

[14] John Greiner. *Semantics-based parallel cost models and their use in provably efficient implementations.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1997.

[15] John Greiner and Guy E. Blelloch. A provably time-efficient parallel implementation of full speculation. *ACM Transactions on Programming Languages and Systems*, 21(2):240–285, 1999.

[16] J. Gustavsson and D. Sands. A foundation for space-safe transformations of call-by-need programs. In *The Third International Workshop on Higher Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1999.

[17] Jon G. Hall, Clem Baker-Finch, Phil Trinder, and David J. King. Towards an operational semantics for a parallel non-strict functional language. In *Proceedings of the International Workshop on the Implementation of Functional Languages (IFL'98)*, LNCS 1595, pages 55–72, 1998.

[18] Matthew Hennessy. *The Semantics of Programming Languages.* Wiley, 1990.

[19] Mark B. Josephs. The semantics of lazy functional languages. *Theoretical Computer Science*, 68:105–111, 1989.

[20] John Launchbury. A natural semantics for lazy evaluation. In *The 20th Symposium on Principles of Programming Languages (POPL'93)*, pages 144–154, 1993.

[21] Hans-Wolfgang Loidl. *Granularity in Large-Scale Parallel Functional Programming.* PhD thesis, Department of Computing Science, University of Glasgow, 1998.

[22] John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317, 1998.

[23] John McCarthy. A basis for a mathematical theory of computation. In *Proc. Western Joint Computer Conference*, pages 225–238, 1961.

[24] Robin Milner. *Communication and Concurrency.* Prentice Hall International, 1989.

[25] Andrew Moran and David Sands. Improvement in a lazy context: An operational theory for call-by-need. In *The 26th Symposium on Principles of Programming Languages (POPL'99)*, pages 45–56, 1999.

[26] Andrew Keith Moran. *Call-by-name, Call-by-need, and McCarthy's Amb.* PhD thesis, Computing Science Department, Chalmers University, 1998.

[27] Luc Moreau. The PCKS-machine: an abstract machine for sound evaluation of parallel functional programs with first-class continuations. In *European Symposium on Programming (ESOP'94)*, LNCS 788, pages 424–438, 1994.

[28] Luc Moreau. The semantics of Scheme with future. In *International Conference on Functional Programming (ICFP'96)*, pages 146–156, 1996.

[29] Chih-Hao Ong. *The Lazy Lambda Calculus.* PhD thesis, Imperial College, 1988.

[30] Simon L. Peyton Jones, John Hughes, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. *Report on the Programming Language Haskell 98: A Non-strict, Purely Functional Language*, 1998. Available from `http://www.haskell.org`.

[31] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

[32] S. Purushothaman and J. Seaman. An adequate operational semantics of sharing in lazy evaluation. In *The 4th European Symposium on Programming (ESOP'92)*, LNCS 582, pages 435–450, 1992.

[33] J. H. Reppy. An operational semantics of first-class synchronous operations. Technical report, Department of Computer Science, Cornell University, 1991.

[34] Paul Roe. *Parallel programming using functional languages.* PhD thesis, Department of Computing Science, University of Glasgow, 1991.

[35] Colin Runciman and David Wakeling. Profiling parallel functional computations (without parallel machines). In *Proceedings of the 1993 Glasgow Workshop on Functional Programming*, pages 236–251, 1994.

[36] David Sands, 1999. Private communication.

[37] Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, 1997.

[38] Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, 1998.

[39] Philip W. Trinder, Kevin Hammond, James S. Mattson, Andrew S. Partridge, and Simon L. Peyton Jones. GUM: A portable implementation of Haskell. In *Proceedings of Programming Language Design and Implementation (PLDI'96)*, pages 79–88, 1996.

[40] Christopher Walton. An abstract machine for memory management. In *Trends in Functional Programming*, chapter 10. Intellect, 2000. To appear.

A Proof of Proposition 12.3

Our proof of Proposition 12.3 mimics the corresponding proof by Launchbury [20]. This appendix contains a more detailed discussion of the proof, much of which comes directly from [20].

We begin by defining in Figure 10 a variant of the denotational semantics that allows us to explicitly deal with finite approximations. The new semantic function takes an extra argument representing a 'resource'. The domain of resources is the least solution of $C = C_\perp$ and we write the lifting injection as S so the elements of C are $\perp, S\perp, SS\perp \ldots$ with limit element ω.

The semantic function $\mathcal{N}$ takes a resource as an extra argument and environments $\sigma : \mathit{Var} \to C \to \mathit{Val}$ now bind variables to functions which take a resource and return a value.

$$
\begin{aligned}
\mathcal{N}[\![e]\!]_\sigma \perp &= \perp \\
\mathcal{N}[\![\lambda x.e]\!]_\sigma (S\ k) &= \mathit{lift}\ \lambda\tau.\mathcal{N}[\![e]\!]_{\sigma[x\mapsto\tau]} \\
\mathcal{N}[\![e\ x]\!]_\sigma (S\ k) &= \mathit{drop}(\mathcal{N}[\![e]\!]_\sigma k)(\mathcal{N}[\![x]\!]_\sigma k) \\
\mathcal{N}[\![x]\!]_\sigma (S\ k) &= \sigma\ x\ k \\
\mathcal{N}[\![\mathbf{let}\ \{x_i = e_i\}_{i=1}^n\ \mathbf{in}\ e]\!]_\sigma (S\ k) &= \mathcal{N}[\![e]\!]_{\mu\sigma'.\sigma[x_i\mapsto\mathcal{N}[\![e_i]\!]_{\sigma'}]_{i=1}^n} k \\
\mathcal{N}[\![e_1\ \mathbf{seq}\ e_2]\!]_\sigma (S\ k) &= \begin{cases} \perp & \text{if } \mathcal{N}[\![e_1]\!]_\sigma k = \perp \\ \mathcal{N}[\![e_2]\!]_\sigma k & \text{otherwise} \end{cases} \\
\mathcal{N}[\![x\ \mathbf{par}\ e]\!]_\sigma &= \mathcal{N}[\![e]\!]_\sigma
\end{aligned}
$$

Figure 10: Resourced denotational semantics

The original semantics is equivalent to $\mathcal{N}$, assuming unlimited resources:

$$\text{If } \forall x.\rho\ x = \sigma\ x\ \omega \text{ then } \quad [\![e]\!]_\rho = \mathcal{N}[\![e]\!]_\sigma \omega$$

By the continuity of $\mathcal{N}$, we have the following lemma:

Lemma A.1
If $[\![e]\!]_\rho \neq \perp$ then there is a natural number m such that $\mathcal{N}[\![e]\!]_\sigma (S^m\ k) \neq \perp$ where $\forall x.\rho\ x = \sigma\ x\ \omega$.

The core result relating the resourced semantics to our operational is the following:

Lemma A.2
If:

$$\mathcal{N}[\![e]\!]_{\mu\sigma'.(x_1\mapsto\mathcal{N}[\![e_1]\!]_{\sigma'},\ldots x_n\mapsto\mathcal{N}[\![e_n]\!]_{\sigma'})}(S^m \perp) \neq \perp$$

then there is a value v, a variable z and a heap H where:

$$\mu\sigma'.(x_1 \mapsto \mathcal{N}[\![e_1]\!]_{\sigma'}, \ldots x_n \mapsto \mathcal{N}[\![e_n]\!]_{\sigma'}) \leq \{\!\{H\}\!\}\rho_0$$

such that:

$$(x_1 \overset{I}{\mapsto} e_1, \ldots x_n \overset{I}{\mapsto} e_n, z \overset{A}{\mapsto} e) \Longrightarrow^* (H, z \overset{A}{\mapsto} v)$$

Proof By induction on m.

As an example of how the proof goes, let $\sigma = \mu\sigma'.(x_1 \mapsto \mathcal{N}[\![e_1]\!]_{\sigma'}, \ldots x_n \mapsto \mathcal{N}[\![e_n]\!]_{\sigma'})$ and consider the case $e = x_i$:

If $\mathcal{N}[\![x_i]\!]_\sigma (S\ k) \neq \perp$ then $\sigma\ x\ k \neq \perp$, so $\mathcal{N}[\![e_i]\!]_\sigma k \neq \perp$. By induction we have:

$$
\begin{aligned}
&(x_1 \overset{I}{\mapsto} e_1, \ldots x_i \overset{A}{\mapsto} e_i, \ldots x_n \overset{I}{\mapsto} e_n, z \overset{I}{\mapsto} x_i) \\
&\qquad\qquad \Longrightarrow^* (H, z \overset{I}{\mapsto} x_i, x_i \overset{A}{\mapsto} v)
\end{aligned}
$$

Using this fact, we can construct the following relation by applying rules block_1 and var:

$$
\begin{aligned}
&(x_1 \overset{I}{\mapsto} e_1, \ldots x_i \overset{I}{\mapsto} e_i, \ldots x_n \overset{I}{\mapsto} e_n, z \overset{A}{\mapsto} x_i) \\
&\quad \Longrightarrow (x_1 \overset{I}{\mapsto} e_1, \ldots x_i \overset{A}{\mapsto} e_i, \ldots x_n \overset{I}{\mapsto} e_n, z \overset{B}{\mapsto} x_i) \\
&\quad \Longrightarrow^* (H, z \overset{B}{\mapsto} x_i, x_i \overset{A}{\mapsto} v) \\
&\quad \Longrightarrow (H', z \overset{A}{\mapsto} \hat{v})
\end{aligned}
$$

■

Finally, the proof of Proposition 12.3 is as follows:

Proof [Proposition 12.3]

By Lemma A.1, if $[\![e]\!]_{\{\!\{x_1\mapsto e_1\ldots x_n\mapsto e_n\}\!\}\rho_0} \neq \perp$ then there is an m such that $\mathcal{N}[\![e]\!]_{\mu\sigma'.(x_i\mapsto\mathcal{N}[\![e_i]\!]_{\sigma'})_{i=1}^n}(S^m\ k) \neq \perp$.

Thus by lemma A.2 we have:

$(x_1 \mapsto e_1 \ldots x_n \mapsto e_n, z \overset{A}{\mapsto} e) \Longrightarrow^* (H, z \overset{I}{\mapsto} v)$ as required. ■