An operational semantics for parallel call-by-need

Contents

An operational semantics for parallel call-by-need

Clem Baker-Finch
University of Canberra*

David J. King
The Open University†

Jon G. Hall
The Open University†

Phil Trinder
Heriot-Watt University‡

Abstract

We present an operational semantics for *parallel call-by-need*, that accurately models the parallel behaviour of a non-strict parallel functional language. Parallelism is modelled synchronously, that is, single reductions are carried out separately, then combined, before proceeding onto the next set of reductions. Consequently the semantics has two levels, with single-step transition rules at one level and combining rules at the other. Parallel threads are modelled by labelled bindings. 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 suitable for proofs. At the same time, it is low level enough to allow us to reason about programs in terms of *parallelism* (i.e. resource usage), as well as *runtime* and *work* done.

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 (Trinder et al. 1996), an established parallel derivative of the non-strict purely functional language Haskell (Peyton Jones et al. 1998). Parallelism is provided in both GPH and the extended λ-calculus in a *mostly-implicit* or *semi-explicit* way, by using the annotation **par** to express parallel composition, whilst leaving thread management to the compiler's runtime system. The semantics presented here is *explicit*, however, 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 *computation* (i.e. what to compute), and *coordination* (i.e. how the computation is arranged in parallel).

A parallel call-by-need semantics is important for theoretical reasons, but there are also several practical applications. As indicated above, 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*; *program X runs faster than program Y.* Being able to prove these properties formally is useful for the ordinary programmer and compiler writer alike. For the programmer,

*School of Computing, University of Canberra, Canberra, Australia. E-mail: `clem@ise.canberra.edu.au`.

†Faculty of Mathematics and Computing, The Open University, Milton Keynes, United Kingdom. E-mail: {`d.j.king, j.g.hall`}`@open.ac.uk`.

‡Department of Computing and Electrical Engineering, Heriot-Watt University, Edinburgh, United Kingdom. E-mail: `trinder@cee.hw.ac.uk`.

they will be able to understand and improve the behaviour of their programs more easily. For the compiler writer, optimising transformations 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 complete re-working of preliminary work presented in Hall et al. (1998), that gave a semantics for the speculative evaluation in a non-determined order of the same language. The semantics presented here has the following key features:

- Bindings are used to model closures, which in turn are used to model threads. Binding labels are used 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).
- It is structured, and relatively simple to use for proof, because it models parallelism synchronously and the rules are at two levels: single-step rules at one level, and multi-step rules at the other level.
- It captures the thread behaviour and evaluation order of an existing parallel functional language implementation, GpH.
- Correctness properties are shown with respect to Launchbury's natural semantics for laziness of a sequential λ-calculus. The proofs are relatively straightforward because our operational semantics is expressed using similar mechanisms to Launchbury (i.e. bindings, heaps, and normalised terms).

2 Related work

There have been several attempts at defining a semantics for call-by-need and call-by-name, for example, Ong (1988), Josephs (1989), Abramsky (1990), Roe (1991), Purushothaman and Seaman (1992), Launchbury (1993), and Ariola et al. (1995). Of these, our parallel call-by-need semantics is perhaps closest to Launchbury's in that we use similar mechanisms such as an explicit heap containing bindings, and normalised terms. Launchbury's semantics does not model parallelism though, and of the aforementioned semantics only the work of Abramsky (1990), Ong (1988), and Roe (1991) give a semantics (in fact a denotational semantics) for a parallel construct. The unique feature of our semantics over others is that we capture the behaviour of parallel threads.

Since the semantics we present captures parallel thread behaviour, it is set at a lower level of abstraction than all of the above semantics. Hence, the semantics of existing parallel and/or concurrent functional languages are in some ways more closely related to our work. There are several languages of this type, some being purely implicit (i.e. languages that use the compiler to generate the parallelism) such as Id, pH, and NESL, as well as languages that are semi-explicit such as Eden and Concurrent ML. Id and pH are quite far removed from our language since they have features such as synchronisation, barriers, and side-effects that need special care in an operational semantics (Aditya et al. 1995, Arvind et al. 1996). Concurrent ML is a distant relation to GpH, since it is a call-by-value language and uses communication primitives. Transition semantics have been given for Concurrent ML (Reppy 1991, Berry et al. 1992) where it is necessary to model processes explicitly. The semantics for Eden and NESL are perhaps more closely related to our work, and so they will be discussed further.

Eden (Breitinger et al. 1998) is a call-by-need parallel and concurrent functional language that provides explicit constructs for specifying the behaviour of processes and channels. The structure of the operational semantics for Eden is similar to our semantics in that it uses a two level transition system — the lower level handling the behaviour of *local* processes and the upper level handling *global* behaviours. The semantics for Eden is set at a lower level than our semantics; that is, it is closer to an abstract machine, and is therefore less amenable to proof. One advantage over our semantics, though, is that communication between processes is modelled, which is an integral part of Eden.

NESL (Blelloch 1995) is a call-by-value functional language, that provides parallelism *implicitly* by implementing several language primitives using nested data-parallelism. GpH does not have special primitives for data-parallelism, although data-parallelism as well as control parallelism can be expressed. Blelloch and Greiner (1996) provide an operational semantics for NESL, and the objectives of their semantics are similar to ours, that is, to provide time and space bounds of programs running on different parallel architectures. The semantics uses an intermediate language with explicit array primitives that provide the parallelism. A DAG (directed acyclic graph) is returned by the semantics where the vertices represent units of computation, and the edges represent control dependencies. A proof in our semantics also implicitly forms a DAG, although no explicit use is made of this.

Our semantics draws on Milner's (1989) work on *process algebras* for concurrency (such as CCS, and SCCS). Indeed, our semantics defines a *synchronous process algebra* where reductions are carried out in lockstep. One of the most well-known ideas in this context is the *bisimulation* relation, which could suggest a means of showing observational equivalences on terms in our language.

3 Modelling parallelism

The operational semantics we present is a two-level transition semantics. At the lower level, there are single-step transition rules for performing reductions on terms, for example, β-reduction. At the upper level, there are rules for combining single-steps (controlling parallelism), and rules for changing the behaviours of terms (task scheduling). Our system owes much to Launchbury's (1993) natural semantics for a sequential call-by-need language, in that we use a heap of bindings to model store and normalised terms to model laziness. Our semantics is set at a lower level of abstraction than Launchbury's, that is, we use a small-step transition system rather a big-step natural semantics. It is widely believed that this level of abstraction is more suitable for modelling parallelism (Hennessy 1990) than that used by Launchbury for a sequential language.

3.1 The extended language (GpH-core)

GpH-core is a simple language used to model the essential parts of 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**.

$$
\begin{array}{rcl}
x & \in & \text{Variable} \\
n & \in & \text{Number} \\
e & \in & \text{Expression} \\
e & ::= & n \\
 & | & x \\
 & | & e_1\ e_2 \\
 & | & \lambda x.e \\
 & | & \mathbf{let}\ x_1 = e_1 \cdots x_n = e_n\ \mathbf{in}\ e \\
 & | & e_1\ \mathbf{seq}\ e_2 \\
 & | & e_1\ \mathbf{par}\ e_2
\end{array}
$$

Following Launchbury (1993), and others, the language is normalised to a restricted syntax. These normalised terms differ from the extended untyped λ-calculus in two ways: all bound variables are distinct; the second argument of application and the first argument of **par** have been replaced with variables. The reason for normalising to this restricted syntax is that it prevents new names being introduced in the transition system, which simplifies the rules. The process of normalisation is straightforward — an algorithm is given in Hall et al. (1998), for example.

$$
\begin{array}{lcl}
x & \in & \text{Variable} \\
n & \in & \text{Number} \\
e & \in & \text{Expression} \\
e & ::= & n \\
 & | & x \\
 & | & e\ x \\
 & | & \lambda x.e \\
 & | & \textbf{let } x_1 = e_1 \cdots x_n = e_n \textbf{ in } e \\
 & | & e_1 \textbf{ seq } e_2 \\
 & | & x \textbf{ par } e
\end{array}
$$

Figure 1: Restricted syntax for GPH-CORE.

Numbers are strictly unnecessary, but 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. This makes the semantics slightly easier, for example, no rule is necessary for the term (x **seq** e) where x is unbound, because this term is not closed.

3.2 Parallel and sequential composition

GPH-CORE expresses parallel coordination using the combinators **par** (for parallel composition) and **seq** (for sequential composition). A denotational semantics of these combinators is defined as follows:

$$
e_1 \textbf{ seq } e_2 = \begin{cases} \perp, & \text{if } e_1 = \perp \\ e_2, & \text{otherwise} \end{cases}, \qquad\qquad e_1 \textbf{ par } e_2 = e_2.
$$

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 operational behaviour of **par** is to evaluate e_1 and e_2 simultaneously, on separate processors, if sufficient resources are available. If there are not sufficient resources then e_1 and e_2 would be evaluated on a single processor. It is important to note that **par** will evaluate e_1 to whnf and no further, hence the necessity for **seq** which is commonly used to force the evaluation of data structures to normal form.

3.3 Heaps and labelled bindings

Following Launchbury (1993) we use a heap H with bindings. In addition to Launchbury, bindings are augmented with labels that indicate thread states. Furthermore, the expression to be reduced (i.e. the focus term in the transition rules) takes the form of a binding. A labelled binding is a mapping from an identifier to a normalised term:

$$
x \overset{\alpha}{\hookrightarrow} e, \qquad \text{where} \quad \alpha ::= \mathit{Inactive} \mid \mathit{Blocked} \mid \mathit{Runnable} \mid \mathit{Active}.
$$

The binding state α will often be abbreviated to the first letter, that is, I, B, R, or A. The binding states are briefly described as follows: a binding is *blocked* if it is waiting for the evaluation of another binding before its evaluation can proceed; it is *runnable* if there are not enough resources, currently available, to evaluate it; and *active* if it is currently being evaluated; all other bindings are in an *inactive* state.

Bindings correspond to heap closures, and in our semantics labelled bindings correspond to threads. In the runtime system for GPH (Trinder et al. 1996), however, 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, although this is not as important for GPH as it is for many other languages. Our semantics zero costs communication, so that an idealised architecture is modelled. Some simulators for GPH, for example, HBC-PP (Runciman and Wakeling 1994), and GRANSIM-Light (Loidl 1998) also model an idealised machine, and they have proven to provide useful information.

Definition (Partitioning the heap)
It is useful to be able to partition the heap, to isolate bindings of a particular type. For this superscripts are used with a binding type, so that:

$$H^\alpha = \{\, x \overset{\alpha}{\mapsto} e \in H \,\}.$$

3.4 Local and global configurations

In the semantics to follow, there are two forms of configuration namely: *local configurations*, and *global configurations*. The *local configuration* is used for describing the evaluation of a single expression on a single processor, and the *global configuration*, for actions applied to all bindings. A local configuration takes the following form:

$$H : x \overset{\alpha}{\mapsto} e.$$

A global configuration is simply a heap H.

Definition (Reduction sequence)
The operational semantics describes a *reduction sequence* from an *initial configuration* to a *final configuration*:

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

where *main* is always used as the program identifier. A notational convention used throughout the semantics, for succinctness, is to use v for values which are whnf expressions in GPH-CORE, that is:

$$\begin{array}{lcl} v & \in & \text{Value} \\ v & ::= & n \mid \lambda x.e \end{array}$$

Note that e in the abstraction could be a redex.

4 Single-step transitions on terms

The single-step transition rules in Figure 2 describe how an active binding should be reduced in the context of a heap. There are several conventions, notational and otherwise, that are used to simplify these rules, and they are best explained by focusing on one rule:

$$(H, x \overset{ABR}{\mapsto} e_1) : z \overset{A}{\mapsto} x \ \mathbf{par}\ e_2 \ \longrightarrow\ (z \overset{A}{\mapsto} e_2)$$

The notation $(H, x \mapsto e)$ is used to partition the heap, separating the binding $(x \mapsto e)$ from all other bindings. The multi-labelled binding for x means a binding where the state is one of A, B, or R, but not I. The right hand side of a single-step transition rule (i.e. $z \overset{A}{\mapsto} e_2$ above) is a heap, but it only includes bindings that are new, or have been changed, so that the binding for x is not lost. The notation $\hat{e}$ is borrowed from Launchbury (1993) and is used to signify a renaming of all variables in the expression e to fresh variable names. The one occurrence of renaming takes place in the variable-rule, which is the only place where names are duplicated, and as Launchbury proves this is sufficient to avoid all unwanted name clashes.

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

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

$$(H, x \overset{ABR}{\mapsto} e) : z \overset{A}{\mapsto} x \longrightarrow (z \overset{B}{\mapsto} x) \qquad \text{(blocking)}$$

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

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

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

$$H : z \overset{A}{\mapsto} \textbf{let } x_1 = e_1 \cdots x_n = e_n \textbf{ in } e \longrightarrow (x_1 \overset{I}{\mapsto} e_1 \cdots x_n \overset{I}{\mapsto} e_n, z \overset{A}{\mapsto} e) \qquad \text{(let)}$$

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

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

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

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

Figure 2: Small-step transition rules for terms.

One of the simplest small-step rules is the **let** rule which is the only rule that populates the heap with new bindings. Note that the only rule that creates more parallelism is the **par** rule. This does not create more parallelism immediately, but instead suggests that a binding should be made active by putting it into an R state, which may be promoted to an A-binding later. This promotion will happen in the multi-step rules, if and only if, enough resources are available. This corresponds to GpH where **par** is thought of as a *hint* of what to parallelise, rather than a *command*.

5 Multi-step transitions on heaps

Multi-step rules apply globally to all the bindings in the heap. They are used to promote bindings into an active state (*activation*); to demote bindings into an inactive state (*deactivation*); to promote blocked bindings to a potentially active state (*unblock*); as well as controlling parallelism (*product*).

5.1 Activation

If sufficient resources are available then a runnable R-binding can be promoted to an active A-binding:

$$\frac{\mid H^A \mid \ < \text{Number of processors}}{(H, x \overset{R}{\mapsto} e) \xrightarrow{act} (H, x \overset{A}{\mapsto} e)} \qquad \text{(activate)}$$

Definition (Maximal activation)
Maximal activation is the continued application of the activate rule, until as many runnable bindings as possible have been made active.

$$(\overset{act}{\Longrightarrow}) = \bigsqcup \xrightarrow{act}{}^{+},$$

where the following ordering is used on heaps:

$$H \sqsubset K \qquad \text{if } \mid H^A \mid \ < \ \mid K^A \mid \ \wedge \ \mathsf{trace}(main, K) \in dom(K^A).$$

this is the maximal relation, and may be thought of as an $\xrightarrow{act}$ normal form. The predicate trace here ensures that the *main* binding, or a binding that *main* needs has priority over other bindings when deciding which ones to promote to an active state. Without trace the *main* binding could be blocked waiting for a sequence of reductions ending with a runnable binding, which may never be chosen, and hence the semantics may not specify a terminating reduction sequence for a terminating program. The predicate trace prevents this, and is defined:

$$\mathsf{trace}(x, K) = \begin{cases} x, & \text{if } x \overset{AR}{\mapsto} e \in K \\ \mathsf{trace}(y, K), & \text{if } x \overset{B}{\mapsto} e^y \in K \end{cases}$$

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

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

Non-deterministic parallel coordination

Non-determinism can occur in the way a program is coordinated in parallel, since there is a choice with maximal activation as to which runnable bindings should be made active. This has no effect on the value computed by a program, that is, the system still satisfies the Church-Rosser Theorem (i.e. confluence is not lost). It does mean, however, that there can be more than one reduction sequence for a given program, but this will only occur when the number of available processors is less than the parallelism described by a program, and doesn't occur for sequential programs.

5.2 Deactivate and unblock

An active or runnable binding that is bound to a whnf term should be made inactive. Moreover, deactivating this binding allows several blocked bindings to potentially become active again. More precisely, all the binding that are immediately blocked waiting for a deactivated binding are moved to a runnable state. Thus,

$$(H, x \overset{AR}{\mapsto} v, z_1 \overset{B}{\mapsto} e_1^x \cdots z_n \overset{B}{\mapsto} e_n^x) \xrightarrow{deact} (H, x \overset{I}{\mapsto} v, z_1 \overset{R}{\mapsto} e_1^x \cdots z_n \overset{R}{\mapsto} e_n^x)$$

Definition (Maximal deactivation)
Maximal deactivation is the continued application of $\stackrel{deact}{\longrightarrow}$ so that all bindings to whnf expressions are made inactive:

$$(\stackrel{deact}{\Longrightarrow}) = \prod \stackrel{deact}{\longrightarrow}+,$$

that is, the minimal relation given the following ordering on heaps:

$$H \sqsubset K \qquad \text{if} \qquad \left| H^{AR} \right| < \left| K^{AR} \right| .$$

The notation $\left| H^{AR} \right|$ corresponds to the number of bindings in an active or runnable state.

5.3 Schedule

Maximal activation and maximal deactivation are now combined into a schedule relation ($\stackrel{sched}{\Longrightarrow}$) that describes the movement of bindings to yield maximum parallelism as described by the language GpH, with respect to the number of processors. The following definition ensures that deactivation takes place before activation, which is a necessary ordering.

$$\frac{H \stackrel{deact}{\Longrightarrow} K \qquad K \stackrel{act}{\Longrightarrow} H'}{H \stackrel{sched}{\Longrightarrow} H'} \quad \text{(schedule)}$$

More consisely schedule can be expressed using relational composition (;), that is,

$$(\stackrel{sched}{\Longrightarrow}) = (\stackrel{deact}{\Longrightarrow}) \, ; (\stackrel{act}{\Longrightarrow}).$$

5.4 Parallel computation

Active bindings are delegated for a single-reduction step by the following product rule, which bears some similarities with Milner's (1989) **Prod** rule. 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:

$$\frac{H^A = \{\, x_i \stackrel{A}{\mapsto} e_i \,\}_{i=1}^{n} \qquad \left\{ H : x_i \stackrel{A}{\mapsto} e_i \longrightarrow H_i \right\}_{i=1}^{n}}{H \stackrel{prod}{\Longrightarrow} H[\bigcup_{i=1}^{n} H_i]} \quad \text{(product)}$$

where H^A is all the active bindings in H, and hence n is the number of active bindings in H. Note that the notation $H : x \stackrel{\alpha}{\mapsto} e$ is defined such that $x \stackrel{\alpha}{\mapsto} e$ is removed from H if it exists. The notation $H[K]$ is used for doing updates, or functional override, that is,

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

5.5 Compute step

Finally, we are in a position to define a parallel compute-step, which applies the product rule followed by a scheduling of bindings. This ordering of rules is important so that all heaps in a reduction sequence have bindings that are in the appropriate state. For example, if this ordering was reversed, then a heap in a reduction sequence could have an active A-binding of a variable to a whnf expression.

$$\frac{H \stackrel{prod}{\Longrightarrow} K \qquad K \stackrel{sched}{\Longrightarrow} H'}{H \Longrightarrow H'} \quad \text{(compute)}$$

Just as with the schedule relation, a compute step can be expressed using relational composition, that is,

$$(\Longrightarrow) = (\overset{prod}{\Longrightarrow})\,;\,(\overset{sched}{\Longrightarrow}).$$

6 Examples

To demonstrate the operational semantics in action we give the reduction sequence for the following program with two processors:

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

Note that the intermediate scheduling phase has been left out of the proof for brevity, so that each heap has been scheduled:

$$\begin{array}{ll}
\{\, main \overset{A}{\mapsto} \mathbf{let}\; f = \lambda x.x, g = (\mathbf{let}\;\; a = 3\;\mathbf{in}\; f\;\; a)\;\mathbf{in}\; g\;\mathbf{par}\;(f\;\; g) \,\} & \\
\Longrightarrow \{\, f \overset{I}{\mapsto} \lambda x.x, g \overset{I}{\mapsto} \mathbf{let}\;\; a = 3\;\mathbf{in}\; f\;\; a, \underline{main \overset{A}{\mapsto} g\;\mathbf{par}\;(f\;\; g)} \,\} & \{\,\text{let}\,\} \\
\Longrightarrow \{\, f \overset{I}{\mapsto} \lambda x.x, \underline{g \overset{A}{\mapsto} \mathbf{let}\;\; a = 3\;\mathbf{in}\; f\;\; a}, \underline{main \overset{A}{\mapsto} f\;\; g} \,\} & \{\,\text{par}\,\} \\
\Longrightarrow \{\, f \overset{I}{\mapsto} \lambda x.x, a \overset{I}{\mapsto} 3, \underline{g \overset{A}{\mapsto} f\;\; a}, \underline{main \overset{A}{\mapsto} (\lambda x.x)\;\; g} \,\} & \{\,\text{let, application}\,\} \\
\Longrightarrow \{\, f \overset{I}{\mapsto} \lambda x.x, a \overset{I}{\mapsto} 3, \underline{g \overset{A}{\mapsto} (\lambda x.x)\;\; a}, \underline{main \overset{A}{\mapsto} g} \,\} & \{\,\text{application, } \beta\text{-reduction}\,\} \\
\Longrightarrow \{\, f \overset{I}{\mapsto} \lambda x.x, a \overset{I}{\mapsto} 3, main \overset{B}{\mapsto} g, \underline{g \overset{A}{\mapsto} a} \,\} & \{\,\beta\text{-reduction, blocking-}I\,\} \\
\Longrightarrow \{\, f \overset{I}{\mapsto} \lambda x.x, a \overset{I}{\mapsto} 3, g \overset{I}{\mapsto} 3, \underline{main \overset{A}{\mapsto} g} \,\} & \{\,\text{variable}\,\} \\
\Longrightarrow \{\, f \overset{I}{\mapsto} \lambda x.x, a \overset{I}{\mapsto} 3, g \overset{I}{\mapsto} 3, main \overset{I}{\mapsto} 3 \,\} & \{\,\text{variable}\,\}
\end{array}$$

6.1 Black holes

Black-holes are detectable infinite loops, and they need special attention in our semantics. Two simple examples of a black-holes are:

$$\begin{array}{l} \mathbf{let}\;\; x = x\;\mathbf{in}\; x \\ \mathbf{let}\;\; x = y, y = z, z = x\;\mathbf{in}\; x \end{array}$$

In our semantics, bindings that depend on the evaluation of others before they are reduced are placed into a blocked state. The most trivial example of this is the binding $(z \overset{B}{\mapsto} z)$, that has a special single-step rule to move it to a blocked state. If the *main* binding is dependent on a black hole, then this will be identified in the semantics by the trace on *main* failing. That is, the value of `trace` will be undefined, since it will not terminate. When this happens, activation will fail, and evaluation stops with failure.

For example,

$$\begin{array}{ll}
\{\, main \overset{A}{\mapsto} \mathbf{let}\;\; x = y, y = x\;\mathbf{in}\; x \,\} & \\
\Longrightarrow \{\, x \overset{I}{\mapsto} y, y \overset{I}{\mapsto} x, main \overset{A}{\mapsto} x \,\} & \{\,\text{let}\,\} \\
\Longrightarrow \{\, x \overset{A}{\mapsto} y, y \overset{I}{\mapsto} x, main \overset{B}{\mapsto} x \,\} & \{\,\text{blocking-}I\,\} \\
\Longrightarrow \{\, x \overset{B}{\mapsto} y, y \overset{A}{\mapsto} x, main \overset{B}{\mapsto} x \,\} & \{\,\text{blocking}\,\}
\end{array}$$

At this point we have a failure, because the compute step will apply the product rule returning:

$$\{\, x \overset{B}{\mapsto} y, y \overset{B}{\mapsto} x, main \overset{B}{\mapsto} x \,\}$$

but now schedule fails, since *main* or a binding that *main* depends on is not active. Our semantics therefore detects black holes.

7 Correctness

Space limitations prevent a full discussion of the correctness of our semantics. In this section we present an overview of the main concepts and results. For full details and a more discursive discussion the reader is referred to Appendix B.

To show that our semantics faithfully describes call-by-need, we take Launchbury's (1993) natural semantics as a standard reference point and establish a congruence between the two definitions. First we specialise our definition to the case of a single processor and show that it corresponds to Launchbury's semantics extended to include **par** and **seq**. Launchbury (1993) also establishes a congruence with a denotational semantics, so we also extend his programme to include **par** and **seq**. Finally, we show that our semantics is deterministic in the sense that the value of an expression is the same for any positive number of processors.

If there is only one processor there is always exactly one active binding, beginning with $main \overset{A}{\mapsto} e$. Furthermore, the blocked bindings will form a chain from that active binding back to *main*. Since the main thread dominates the processor the R-bindings are redundant and the activation rule can be combined with the relevant single-step rules.

At most one deactivation will be necessary following every computation step so it can be treated as just another single-step rule. The product rule is redundant since there is only ever one active binding. The single-processor rules can thus be derived case by case. The new transition relation is denoted $\underset{1}{\Longrightarrow}$.

Launchbury's evaluation relation is written $\Gamma : e \Downarrow \Delta : v$ where Γ and Δ are heaps similar to ours except that bindings do not carry labels. The reader should refer to Launchbury (1993) for a more detailed discussion. It is straightforward to extend his semantics to include **par** and **seq**.

To relate the two semantics we will need to erase binding labels in one direction and add labels in the other direction. It is sufficient, for our purposes, to label all bindings as being *Inactive* since labels only reflect information about the computation that has gone before and have no influence on the evaluation of the current expression. If H is a heap then $\overline{H}$ is an unlabelled heap consisting of the same bindings. If Γ is an unlabelled heap then Γ^I is a heap with the same bindings, all labelled *Inactive*. Both functions can be defined inductively.

We are now in a position to prove that any expression e reduces to a whnf v in our semantics if and only if e evaluates to v in Launchbury's semantics.

Theorem 1
If $\Gamma : e \Downarrow \Delta : v$ then $(\Gamma^I, z \overset{A}{\mapsto} e) \underset{1}{\Longrightarrow}^* (\Delta^I, z \overset{A}{\mapsto} v)$ for some fresh variable z.

Sketch Proof By induction on the length of the natural semantics derivation. □

Theorem 2
If $(H, z \overset{A}{\mapsto} e) \underset{1}{\Longrightarrow}^* (H', z \overset{A}{\mapsto} v)$ then $\overline{H} : e \Downarrow \overline{H}' : v$ for some fresh variable z.

Sketch Proof By induction on the length of the reduction $\underset{1}{\Longrightarrow}^*$. □

The extension of Launchbury's congruence with the denotational semantics is straightforward. A more complete description of the whole proof can be found in Appendix B.

8 Comparing programs in terms of time and parallelism

One of the 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 apply standard metrics for parallelism, work done, efficiency, and speedup (see Eager et al. (1989) 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 runtime. In fact, there are some simulators, such as HBC-PP (Runciman and Wakeling 1994) that count similar reductions to those in our semantics, and it has been found to give useful performance measurements.

Definition (Work done and runtime)
Given a reduction sequence $H_0 \implies H_1 \implies \cdots \implies H_{t_n}$ for a terminating program with n processors, then the total work done, and runtime with respect to n processors are given by:

$$W(n) = \sum_{i=0}^{t_n} \mid H_i^A \mid , \qquad R(n) = t_n.$$

Definition (Average parallelism)
Given an unbounded number of processors, the average parallelism, or speedup is given by:

$$\text{Average parallelism} = \frac{W(\infty)}{t_\infty}, \qquad P(n) = \frac{W(n)}{t_n}.$$

In practice it is useful to use a more general definition (P) that takes into account the number of processors.

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

$$\text{Maximum parallelism} = \max\{ \mid H_i^A \mid \}_{i=0}^{t_\infty}, \qquad M(n) = \max\{ \mid H_i^A \mid \}_{i=0}^{t_n}.$$

Again in practice it is useful to use a more general definition of maximum parallelism (M), that takes into account the number of processors.

For the example, given the in Section 6, with a single processor and two processors we have:

	1 processor	2 processors
Runtime	10	7
Maximum parallelism w.r.t. n	1	2
Work done	10	10
Average parallelism w.r.t. n	1.0	$1\frac{3}{7}$

This demonstrates that the given program can make use of more than one processor to improve its runtime.

Definition (Speedup and Efficiency)
Given a reduction sequence $H_0 \implies H_1 \implies \cdots \implies H_{t_n}$ for a terminating program with n processors, speedup and efficiency on n processors are given by:

$$S(n) = \frac{t_1}{t_n}, \qquad E(n) = \frac{S(n)}{n}$$

9 How close is the semantics to the GpH runtime system?

One way of summarising both the semantics and the GpH runtime system is to express the state changes made by threads in GpH and by bindings in the semantics.

9.1 GpH runtime system

Figure 3 describes what states a thread may change to as defined by the GUM (Trinder et al. 1996) runtime system for GpH. There is always a main thread that represents a program, but all other threads are introduced by the **par** combinator. With GUM, the **par** combinator initially labels the heap closure that represent the computation of its first argument as a *spark*. Sparking is a commonly used mechanism for generating threads, see Clack and Peyton Jones (1986). All the heap closures that are sparks are placed into a spark pool, which represents the closures that could be evaluated as threads. When a PE is available then it looks for work to schedule from its spark pool. If the chosen spark is not in whnf, then it is evaluated on the PE as a thread (i.e. made *active*). Otherwise, if a spark turns out to represent a value (i.e. whnf) then it is discarded (i.e. made *inactive*) If a PE is available, but there are no local sparks in the spark pool that represent non-whnf expressions, then work from another PE's spark pool is sought by sending a fishing message that "swims" from PE to PE looking for work (i.e. a closure is *fetched*). If a thread T cannot continue evaluating a closure because it depends on another thread S, then thread S is placed into the *blocking queue* of thread T (i.e. S *blocks*). When a thread completes (i.e. the closure is reduced to whnf), then the closure is overwritten with the value, and all the threads that are blocked on this closure become *runnable*.

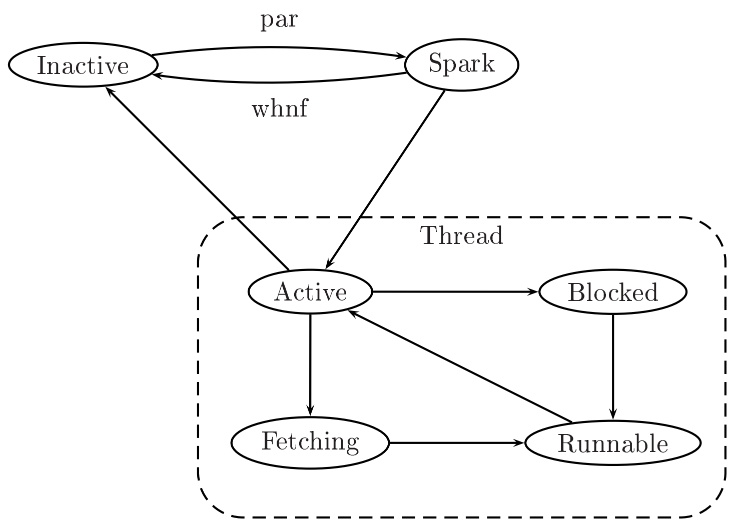

Figure 3: A state transition diagram for threads in GpH.

9.2 Binding state transitions in the semantics

In the small-step transition rules bindings are reduced, therefore most of the small-step rules do not change the state of a binding, but instead change the expression part of the binding. The binding state changes that are made in the small-step rules are from *Active* to *Blocked* (by the rules blocking-I, blocking, black hole, application, and seq); and from *Inactive* to *Runnable* (by the rules blocking-I, and par). All the other binding state changes are made by the scheduling rules. The complete picture of binding state transtions made by a compute step is given in Figure 4.

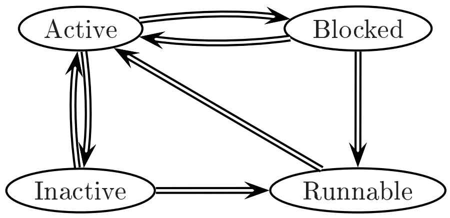

Figure 4: A state transition diagram for a compute step in the semantics.

9.3 What are the differencs?

The two most obvious differences are that the GPH runtime system uses sparks as an intermediate structure for implementation, and that communicatation (i.e. fetching) takes place. Further, for simplicity in the semantics we think of all bindings as threads that are in a particular state, so that all heap closures have a thread state. This is quite different from the runtime system where there is no such thing as an *Inactive* thread, but instead these are just heap closures.

Without the above differences there are two extra state transitions in the semantics:

1. The transition $I \implies R$ also occurs in the GPH runtime system via an inactive closure becoming a spark, before being made active. Thus the spark-state in Figure 3 can be thought of as a *Runnable* state.
2. The transition $B \implies A$ happens in two stages in the semantics, i.e. $B \overset{deact}{\Longrightarrow} R$ followed by $R \overset{act}{\Longrightarrow} A$, although the intermediate stage is not shown in the final compute step.

10 Applications of the semantics

The semantics is not just of theoretical importance, but it has several useful applications, which we will now briefly describe.

10.1 Equational reasoning for parallel coordination

Functional languages are famous for being good at equational reasoning, that is, properties of programs are easily shown by using formal techniques. Using equational reasoning to show the properties parallel programs is much more difficult, though, because parallel programs express *coordination* as well as *computation*. In Section 8 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:

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

Once we have an equational system for **par** and **seq** the next step is to have a system for higher-order parallel coordination functions (these are known as evaluation strategies (Trinder et al. 1998)). Our current approach for comparing parallel programs is quite straightforward, and hasn't been fully explored. It may worth exploring more abstract techniques such as Moran and Sands (1999) did with their operational theory for call-by-need, or bisimulation (Milner 1989).

10.2 Abstract machines

We would like to use the semantics to derive an abstract machine for lazy parallel graph reduction, just as Sestoft (1997) did for Launchbury's sequential lazy semantics. We have already developed an interpreter for our semantics, that allows us to analyse the behaviour of an expression automatically. This has been extremely useful in studying examples that can be tedious to work out by hand. The abstract machine will be derived from our interpreter, and it will allow us to study more closely the behaviour of a real compiler. It may also be possible to use the semantics, at this level, to formally justify the behaviour of parallel simulators, such as HBC-PP (Runciman and Wakeling 1994), or GranSim (Loidl 1998).

10.3 Profiling

There currently exist two methods of execution profiling for GpH, both of which involve adding annotations to a program (Hammond et al. 1997, King et al. 1998). The semantics for the two methods is quite different — they use different scoping rules, and annotate different behaviours. These differences, and perhaps similarities can be formally shown with our semantics.

11 Conclusions and further work

11.1 Conclusions

In this paper we have given an operational semantics for parallel call-by-need that models the language GpH. The semantics uses the mechanism of a heap of labelled bindings to model sharing, and parallel thread behaviour. The resulting semantics is much simpler than many other attempts at a parallel semantics. We were also able to show the correctness of our semantics with respect to Launchbury's sequential semantics since we share a number of key features.

The semantics is quite general and could easily be refined to model other behaviours such as: sequential evaluation (by restricting to one processor); speculative evaluation (by replacing I and B-bindings with R-bindings); or used to model a machine with unlimited resources.

11.2 Extending the semantics

Apart from extending the semantics to model other terms such as case and constructors, there are two main shortcomings of the semantics, that provide avenues for future work, namely, we would like to model communication, and space usage more accurately.

Communication. 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 runtime.

Space. The next extension which is related to modelling communication is to model space usage more accurately. Currently we have a simple model that essentially uses one large heap space for all the storage space. This doesn't match parallel compilers, that typically distribute information across processors. For instance, GpH uses a separate heap on each processor, and communicates threads between heaps.

References

Abramsky, S. (1990), The lazy lambda calculus, *in* D. Turner, ed., Research Topics in Functional Programming, Addison Wesley, pp. 65–117.

Aditya, S., Arvind, Augustsson, L., Maessen, J.-W. and Nikhil, R. S. (1995), Semantics of pH: A parallel dialect of Haskell, *in* P. Hudak, ed., Proceedings of the Haskell Workshop, La Jolla, California, pp. 35–49.
URL: `ftp://csg-ftp.lcs.mit.edu/pub/papers/csgmemo/memo-376.ps.gz`

Ariola, Z., Felleisen, M., Maraist, J., Odersky, M. and Wadler, P. (1995), A call-by-need lambda calculus, *in* The 22'nd Symposium on Principles of Programming Languages (POPL'95), ACM SIGPLAN-SIGACT, San Francisco, California, pp. 233–246. See also (Maraist et al. 1998).
URL: `http://cm.bell-labs.com/cm/cs/who/wadler/topics/call-by-need.html`

Arvind, Maessen, J.-W., Nikhil, R. S. and Stoy, J. E. (1996), λ_s an implicitly parallel λ-calculus with letrec, synchronization and side-effects, memo 393, MIT Laboratory for Computer Science, Cambridge, Massachusetts.
URL: `ftp://csg-ftp.lcs.mit.edu/pub/papers/csgmemo/memo-393.ps.gz`

Berry, D., Milner, R. and Turner, D. N. (1992), A semantics for ML concurrency primitives, *in* The 19'th Symposium on Principles of Programming Languages (POPL'92), ACM.
URL: `http://www.dcs.gla.ac.uk/fp/authors/David_N_Turner/`

Blelloch, G. E. (1995), *NESL: A Nested Data-Parallel Language (Version 3.1)*, Pittsburgh, PA 15213.
URL: `http://www.cs.cmu.edu/~scandal/papers.html`

Blelloch, G. E. and Greiner, J. (1996), A provable time and space efficient implementation of NESL, *in* International Conference on Functional Programming (ICFP'96), pp. 213–225.
URL: `http://www.cs.cmu.edu/~scandal/papers.html`

Breitinger, S., Loogen, R., Ortega-Mallén, Y. and Peña, R. (1998), *Eden: Language Definition and Operational Semantics*, Fachbereich Mathematik und Informatik, Philipps Universität Marburg. Technical Report 96-10.
URL: `http://www.mathematik.uni-marburg.de/~breiting/eden.html`

Clack, C. and Peyton Jones, S. L. (1986), The four-stroke reduction engine, *in* Conference on Lisp and Functional Programming (LFP'86), ACM Press, Cambridge, Massachusetts.

Eager, D. L., Zahorhan, J. and Lazowska, E. D. (1989), Speedup versus efficiency in parallel systems, *IEEE Transactions on Computers* **38**(3), 408–423.

Hall, J. G., Baker-Finch, C., Trinder, P. and King, D. J. (1998), 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, Springer-Verlag, London, pp. 55–72.
URL: `http://www.cee.hw.ac.uk/~dsg/gph/papers/`

Hammond, K., Loidl, H.-W. and Trinder, P. (1997), Parallel cost centre profiling, *in* Proceedings of the Glasgow Workshop on Functional Programming, Ullapool, Scotland.

Hennessy, M. (1990), *The Semantics of Programming Languages*, Wiley.

Josephs, M. B. (1989), The semantics of lazy functional languages, *Theoretical Computer Science* **68**, 105–111.

King, D. J., Hall, J. G. and Trinder, P. (1998), A strategic profiler for Glasgow Parallel Haskell, *in* Proceedings of the International Workshop on the Implementation of Functional Languages (IFL'98), LNCS 1595, Springer-Verlag, London, pp. 90–104.
URL: `http://www.cee.hw.ac.uk/~dsg/gph/papers/`

Launchbury, J. (1993), A natural semantics for lazy evaluation, *in* The 20'th Symposium on Principles of Programming Languages (POPL'93), ACM SIGPLAN-SIGACT, Charleston, South Carolina, pp. 144–154.
URL: `http://www.cse.ogi.edu/~jl/Papers/lazySem.ps`

Loidl, H.-W. (1998), Granularity in Large-Scale Parallel Functional Programming, PhD thesis, Department of Computing Science, University of Glasgow.

Maraist, J., Odersky, M. and Wadler, P. (1998), A call-by-need lambda calculus, *Journal of Functional Programming* **8**(3), 275–317.
URL: `http://cm.bell-labs.com/cm/cs/who/wadler/topics/call-by-need.html`

Milner, R. (1989), *Communication and Concurrency*, Prentice Hall International.

Moran, A. and Sands, D. (1999), Improvement in a lazy context: An operational theory for call-by-need, *in* The 26th Symposium on Principles of Programming Languages (POPL'99), ACM SIGPLAN-SIGACT, ACM Press, San Antonio, Texas, pp. 45–56.
URL: `http://www.cs.chalmers.se/~andrew/ftp/cbneed-theory.ps.gz`

Ong, C.-H. (1988), The Lazy Lambda Calculus, PhD thesis, Imperial College, London.

Peyton Jones, S. L., Hughes, J., Augustsson, L., Barton, D., Boutel, B., Burton, W., Fasel, J., Hammond, K., Hinze, R., Hudak, P., Johnsson, T., Jones, M., Launchbury, J., Meijer, E., Peterson, J., Reid, A., Runciman, C. and Wadler, P. (1998), *Report on the Programming Language Haskell 98: A Non-strict, Purely Functional Language*. Available from `http://www.haskell.org`.
URL: `http://www.haskell.org`

Purushothaman, S. and Seaman, J. (1992), An adequate operational semantics of sharing in lazy evaluation, *in* The 4'th European Symposium on Programming (ESOP'92), LNCS 582, Springer-Verlag, Rennes.

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

Roe, P. (1991), Parallel programming using functional languages, PhD thesis, Department of Computing Science, University of Glasgow, Glasgow, Scotland.
URL: `http://sky.fit.qut.edu.au/~proe/papers.html`

Runciman, C. and Wakeling, D. (1994), Profiling parallel functional computations (without parallel machines), *in* J. T. O'Donnell and K. Hammond, eds, Proceedings of the 1993 Glasgow Workshop on Functional Programming, Springer-Verlag, Ayr, Scotland, pp. 236–251.

Sestoft, P. (1997), Deriving a lazy abstract machine, *Journal of Functional Programming* **7**(3), 231–264.

Trinder, P. W., Hammond, K., Loidl, H.-W. and Peyton Jones, S. L. (1998), Algorithm + Strategy = Parallelism, *Journal of Functional Programming* **8**(1), 23–60.
URL: `http://www.dcs.gla.ac.uk/~hwloidl/publications/strategies.ps.gz`

Trinder, P. W., Hammond, K., Mattson, J. S., Partridge, A. S. and Peyton Jones, S. L. (1996), GUM: A portable implementation of Haskell, *in* Proceedings of Programming Language Design and Implementation (PLDI'96), Philadelphia, USA.
URL: `http://www.dcs.gla.ac.uk/fp/authors/Philip_Trinder/gumFinal.ps.Z`

A An interpreter

What follows is an interpreter for GpH-core written in Haskell, that shows the reduction sequence of terms according to the operational semantics given in this paper.

```
data Exp = Con Int
         | Var Name
         | Let [LetBinding] Exp
         | Abs Name Exp
         | App Exp Exp
         | Par Exp Exp
         | Seq Exp Exp
      deriving (Show, Eq)

type Name = String

data LetBinding = Name := Exp
      deriving (Show, Eq)

data ThreadState = Inactive | Runnable | Active | Blocked
      deriving (Eq, Ord, Show)

type Binding = (Name, ThreadState, Exp)

type Heap = [Binding]

name :: Binding -> Name
name (n,_,_) = n

state :: Binding -> ThreadState
state (_,t,_) = t

term :: Binding -> Exp
term (_,_,e) = e
```

Figure 5: Syntax and datatypes.

```
type S a = State -> (a, State)
type State = Int

returnS :: a -> S a
returnS a s = (a, s)

bindS :: S a -> (a -> S b) -> S b
bindS m k s = k a s1
   where
      (a, s1) = m s

freshId :: S Exp
freshId = \s -> (Var ("x" ++ show s), s+1)

mapS :: (a -> S b) -> [a] -> S [b]
mapS f []     = returnS []
mapS f (x:xs) = f x           `bindS` \fx ->
                mapS f xs     `bindS` \fxs ->
                returnS (fx: fxs)

norm :: Exp -> S Exp
norm (Con n)                  = returnS (Con n)
norm (Var x)                  = returnS (Var x)
norm (Let bs e)               = mapS (\(x := e1) -> norm e1) bs
                                               `bindS`         \bs' ->
                                 norm e        `bindS`         \e' ->
                                 returnS (Let
                                   (zipWith (:=) (map (\(b:=e2)->b) bs) bs') e')
norm (Abs x e)                = norm e        `bindS`         \e' ->
                                 returnS (Abs x e')
norm (App e (Var x))          = returnS (App e (Var x))
norm (App e1 e2)              = norm e2       `bindS`         \e2' ->
                                 freshId       `bindS`         \(Var x) ->
                                 returnS (Let [x := e2']
                                              (App e1 (Var x)))
norm (Par (Var x) e)          = norm e        `bindS`         \e' ->
                                 returnS (Par (Var x) e')
norm (Par e1 e2)              = norm e1       `bindS`         \e1' ->
                                 freshId       `bindS`         \(Var x) ->
                                 norm e2       `bindS`         \e2' ->
                                 returnS (Let [x := e1'] (Par (Var x) e2'))
norm (Seq e1 e2)              = returnS (Seq e1 e2)
norm x                        = error ("No rule to normalise: " ++ show x)

normalise :: Exp -> Exp
normalise e = e' where (e', s') = norm e 0
```

Figure 6: Normalisation.

```
reduce :: Binding -> Heap -> Heap

reduce (z, Active, Var x) hs | z==x                        -- BLACK HOLE
      =  [(z, Blocked, Var z)]

reduce (z, Active, Var x) hs | if_whnf                     -- VARIABLE
      = case lookup of Just v -> [(z, Active, v)]
      where
      lookup = lookupBinding x hs
      if_whnf = case lookup of
                  Nothing -> error "reduce(var) -- non-closed term"
                  Just v  -> whnf v ((z, Active, Var x):hs)

reduce (z, Active, Var x) hs                               -- BLOCKING
      = case (findBinding x hs) of
          Nothing                 -> error "reduce(var-blocking)"
          Just ((Inactive,e), hs') -> [(z, Blocked, Var x), (x, Runnable, e)]
          Just _                  -> [(z, Blocked, Var x)]

reduce (z, Active, App (Abs y e) (Var x)) hs               -- BETA-REDUCTION
      = [(z, Active, subst x y e)]

reduce (z, Active, App e (Var x)) hs                       -- APPLICAION
      = case (findBinding z (reduce (z, Active, e) hs)) of
          Nothing           -> error "reduce(app)"
          Just ((s,e'), hs') -> (z, s, App e' (Var x)): hs'

reduce (z, Active, Let bs e) hs                            -- LET
      = (z, Active, e): map upgradeBinding bs

reduce (z, Active, Seq v e) hs | whnf v hs                 -- SEQ-ELIM
      = [(z, Active, e)]

reduce (z, Active, Seq e1 e2) hs                           -- SEQ
      = case (findBinding z (reduce (z, Active, e1) hs)) of
          Nothing             -> error "reduce(seq)"
          Just ((s, e1'), hs') -> (z, s, Seq e1' e2) : hs'

reduce (z, Active, Par (Var x) e2) hs                      -- PAR
      = case (findBinding x hs) of
          Nothing                 -> error "reduce(par)"
          Just ((Inactive,e1),hs') -> [(z, Active, e2), (x, Runnable, e1)]
          Just _                  -> [(z, Active, e2)]
```

Figure 7: Single-step reductions.

```
schedule :: Int -> Heap -> Heap
schedule np t = activate np (deactivate t)

activate :: Int -> Heap -> Heap
activate np hs = is ++ bs ++ rs' ++ as'
      where
      hs' = traceMain "main" hs []
      (is,bs,rs,as) = partitionHeap hs'
      (potential_as, rs') = splitAt (np - length as) rs
      as' = (map (toState Active) potential_as) ++ as

traceMain :: String -> Heap -> [String] -> Heap
traceMain x hs visited
      = case (findBinding x hs) of
          Nothing -> error "traceMain -- Unbound variable"
          Just ((s,e), hs') ->
               if (s==Active) || (s==Runnable)
                 then (x,s,e) : hs'
                 else if (s==Blocked)
                        then if x `elem` visited
                               then error "traceMain -- Possible blackhole"
                               else traceMain (blocker e) hs (x:visited)
                        else hs

blocker :: Exp -> String
blocker (Var x) = x
blocker (Seq (Var x) e) = x
blocker (App (Var x) e) = x
blocker _ = error "blocer -- something is blocked that shouldn't be"

deactivate :: Heap -> Heap
deactivate hs = (is ++ new_is) ++ stay_bs ++ new_rs ++ stay_rs_as
      where
      (is, bs, rs, as) = partitionHeap hs
      (whnfs, stay_rs_as) = partition (\b -> whnf (term b) hs) (rs++as)
      (potential_rs, stay_bs) = partition unblock bs
      new_rs = map (toState Runnable) potential_rs
      new_is = map (toState Inactive) whnfs

      unblock (z,s,e) = or [ blocked_on e x | (x,s',e') <- whnfs ]

blocked_on :: Exp -> String -> Bool
blocked_on e x = case e of
                      Var y           -> x==y
                      Seq (Var y) e   -> x==y
                      (App (Var y) e) -> x==y
                      _               -> False
```

Figure 8: Scheduling.

```
init_config :: Exp -> Heap
init_config e = [("main", Active, e)]

is_final_config :: Heap -> Bool
is_final_config hs = case (getBinding "main" hs) of
                       Nothing -> error "no main"
                       Just e  -> (state e) == Inactive

prod :: Heap -> Heap
prod hs = hs `functional_override` (bigUnionBindings hss)
      where
      hss = map reduce_term as
      reduce_term b = reduce b (removeBinding (name b) hs)
      as = filter isActive hs
      isActive b = state b == Active

step :: Int -> Heap -> Heap
step np hs = schedule np (prod hs)

steps :: Int -> Heap -> [Heap]
steps np hs = iterate (step np) hs

reduction_sequence :: Int -> Exp -> [Heap]
reduction_sequence np e = takeWhile' (not . is_final_config) run_forever
      where
      run_forever = steps np (init_config (normalise e))
      takeWhile' p []                = []
      takeWhile' p (x:xs) | p x       = x:takeWhile' p xs
                          | otherwise = [x]
```

Figure 9: A reduction sequence.

```
findBinding :: Name -> Heap -> Maybe ((ThreadState, Exp), Heap)
findBinding x bs
      = case (partition (eqName x) bs) of
          ([],  bs') -> Nothing
          ([b], bs') -> Just ((state b, term b), bs')
          (_,   bs') -> error ("findBindingIn; Multiple occurrences of "
                                ++ x)

eqName :: Name -> Binding -> Bool
eqName x b = x == name b

lookupBinding :: Name -> Heap -> Maybe Exp
lookupBinding x bs = case (getBinding x bs) of
                       Nothing -> Nothing
                       Just b  -> Just (term b)

getBinding :: Name -> Heap -> Maybe Binding
getBinding x bs
      = case [ b | b <- bs, x == name b] of
          []  -> Nothing
          [a] -> Just a
          _   -> error ("getBinding; Multiple occurrences of " ++ x)

upgradeBinding :: LetBinding -> Binding
upgradeBinding (x := e) = (x, Inactive, e)

toState :: ThreadState -> Binding -> Binding
toState s' (x, s, e) = (x, s', e)

unionHeap :: Heap -> Heap -> Heap
as `unionHeap` bs  = nub (as ++ bs)

bigUnionBindings :: [Heap] -> Heap
bigUnionBindings bss = foldr unionHeap [] bss

removeBinding :: Name -> Heap -> Heap
removeBinding n bs = [ b | b <- bs, name b /= n ]

functional_override :: Heap -> Heap -> Heap
xs `functional_override` ys
       = [ x  | x <- xs, not (name x `elem` names_ys)] ++ ys
      where
      names_ys = map name ys
```

Figure 10: Operations on heaps and bindings.

```
whnf :: Exp -> Heap -> Bool
whnf (Con n) hs               = True
whnf (Var x) hs               = (lookupBinding x hs) == Nothing
whnf (Let bs e) hs            = False
whnf (Abs x e) hs             = True
whnf (App e (Var x)) hs       = False
whnf (Seq e1 e2) hs           = False
whnf (Par (Var x) e) hs       = False
whnf e hs                     = error ("whnf -- no case for" ++ show e)

subst :: Name -> Name -> Exp -> Exp
subst x y (Con n)             = Con n
subst x y (Var w) | w==y      = Var x
                  | otherwise = Var w
subst x y (Let bs e)          = Let [ w := subst x y e1 | w:=e1 <- bs]
                                    (subst x y e)
subst x y (Abs w e)           = Abs w (subst x y e)
subst x y (App e1 e2)         = App (subst x y e1) (subst x y e2)
subst x y (Par e1 e2)         = Par (subst x y e1) (subst x y e2)
subst x y (Seq e1 e2)         = Seq (subst x y e1) (subst x y e2)
subst x y e                   = error ("subst -- no case for" ++ show e)
```

Figure 11: The whnf predicate and variable substitution.

```
runtime :: [Heap] -> Int
runtime hss = (length hss) - 1

max_par :: [Heap] -> Int
max_par hss = maximum (map (length . filter isActive) hss)

work_done :: [Heap] -> Int
work_done hss = sum (map (length . filter isActive) hss)

average_par :: [Heap] -> Float
average_par hss = fromInt (sum (map (length . filter isActive) hss))
                      / fromInt (runtime hss)
```

Figure 12: The metrics.

B A more detailed correctness proof

A valid and sensible approach to demonstrating the correctness of our semantics is as follows:

- Restrict our semantics to the case of a single processor and show that it corresponds to Launchbury's semantics, extended to include **par** and **seq**.
- Extend Launchbury's proof of correctness with respect to the denotational semantics.
- Show that our semantics produces the same result (i.e. value bound to `main`) no matter how many processors are available.

This paper follows the program through in detail.

Deriving the 1-processor case

We begin by deriving a version of our semantics that is specialised to the case of a single processor, thus simplifying the proof of correspondence with Launchbury's natural semantics.

The aim is to eliminate the 'two-level' nature of the rules by taking advantage of the observation that the product rule is trivial in the 1-processor case and by deriving scheduling rules that can be treated as alternative computation rules.

Runnable, blocked and active bindings

Clearly, if if there is only one processor, there is always exactly one active binding (see the activation rule) starting with $\texttt{main} \overset{A}{\mapsto} e$ and ending with $\texttt{main} \overset{A}{\mapsto} e$.

In that case there is no longer any need for the *runnable* category of binding. Inspecting the single-step rules, R-bindings may be introduced by a **par** but since they are not on the main thread they will never be activated and may as well remain as I-bindings. Runnable bindings also arise via the $block_1$ rule in which case they are on the main thread, since they follow from the single active binding. Similarly, when the active expression reaches whnf a blocked binding can be made runnable by the deactivation phase. In either case, the R-binding will immediately be activated by the following activation phase. Since that is the only purpose of the activation rule we can combine it with the 1-processor version of the $block_1$ rule (see below) thus eliminating the need for R-bindings at all.

Blocked bindings are only constructed by the $block_1$ and *black-hole* rules. It is clear that all blocked computations can be traced forward to the current active computation (or black hole).

Summarising, in the 1-processor case there is no need for runnable bindings, there will always be exactly one active binding and the blocked bindings will form a chain from that active binding back to `main`. There will be no other blocked bindings.

Scheduling rules

As has already been mentioned, the activation rule is to be eliminated by combining it with the $block_1$ rule and with the deactivation rule, to which we now turn our attention.

The deactivation phase is simplified by noting that at most one deactivation will be necessary following every computation step. Hence it can be written on the same level as the computation rules. Its effect will be to deactivate $z \overset{A}{\mapsto} v$ and pass control directly back to the binding that is blocked on z:

$$(H, x \overset{B}{\mapsto} e^{(z)}, z \overset{A}{\mapsto} v) \underset{1}{\Longrightarrow} (H, z \overset{I}{\mapsto} v, x \overset{A}{\mapsto} e^{(z)}) \qquad (val)$$

Computation rules

The *prod* rule becomes redundant since there is only ever one active binding. That is, it is only used to combine singleton sets of computation steps. In the remainder of this section we will justify each 1-processor rule in turn, by derivation from the more general case. The new transition relation will be denoted $\underset{1}{\Longrightarrow}$.

let

Combining *let* with *prod* we get:

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

So we can write the 1-processor rule as:

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

var

Combining *var* with *prod* gives:

$$\frac{(H, x \overset{I}{\mapsto} v) : z \overset{A}{\mapsto} x \ \longrightarrow\ (z \overset{A}{\mapsto} \widehat{v})}{(H, x \overset{I}{\mapsto} v, z \overset{A}{\mapsto} x) \overset{prod}{\Longrightarrow} (H, x \overset{I}{\mapsto} v, z \overset{A}{\mapsto} \widehat{v})}$$

justifying:

$$(H, x \overset{I}{\mapsto} v, z \overset{A}{\mapsto} x) \underset{1}{\Longrightarrow} (H, x \overset{I}{\mapsto} v, z \overset{A}{\mapsto} \widehat{v}) \qquad (var\text{-}elim)$$

Note on a potentially confusing change of notation: The *var* rule neatly pairs with $block_1$ in the same way that we have *app, seq, par* and *app-elim, seq-elim, par-elim* rules, so I have renamed *var* as *var-elim* and $block_1$ becomes *var* below. I hope that's not too confusing ...

block$_1$

Combining $block_1$ with *prod* we get:

$$\frac{(H, x \overset{RI}{\mapsto} e) : z \overset{A}{\mapsto} x \ \longrightarrow\ (z \overset{B}{\mapsto} x, x \overset{R}{\mapsto} e)}{(H, x \overset{RI}{\mapsto} e, z \overset{A}{\mapsto} x) \overset{prod}{\Longrightarrow} (H, z \overset{B}{\mapsto} x, x \overset{R}{\mapsto} e)}$$

If z is active then with a single processor z must be the point of the main trace. If z becomes blocked on x, clearly x is now on the main trace and will thus be activated by the immediately following schedule rule. Hence we activate x directly in the *var* rule and the activation part of the scheduling becomes redundant. Since there is no need to distinguish runnable bindings, we can delete R from the left hand side of the transition:

$$(H, x \overset{I}{\mapsto} e, z \overset{A}{\mapsto} x) \underset{1}{\Longrightarrow} (H, z \overset{B}{\mapsto} x, x \overset{A}{\mapsto} e) \qquad (var)$$

block$_2$

The *block$_2$* rule becomes redundant. If z is blocked on x and x is also blocked then we have a black hole (since x must be on the main thread). Removing this rule makes the computation stop, thus detecting the black hole.

black-hole

Similarly, the *black-hole* rule is redundant in the 1-processor case. Removing this rule means there will be no transition from $z \overset{A}{\mapsto} z$ so the computation will stop, again detecting the black hole.

app

No change is necessary:

$$\frac{\dfrac{H : z \overset{A}{\mapsto} e \;\longrightarrow\; (H', z \overset{\alpha}{\mapsto} e')}{H : z \overset{A}{\mapsto} e\, y \;\longrightarrow\; (H', z \overset{\alpha}{\mapsto} e'\, y)}}{(H, z \overset{A}{\mapsto} e\, y) \overset{prod}{\Longrightarrow} (H', z \overset{\alpha}{\mapsto} e'\, y)}$$

Gives the following 1-processor version:

$$\frac{(H, z \overset{A}{\mapsto} e) \underset{1}{\Longrightarrow} (H', z \overset{\alpha}{\mapsto} e')}{(H, z \overset{A}{\mapsto} e\, y) \underset{1}{\Longrightarrow} (H', z \overset{\alpha}{\mapsto} e'\, y)} \qquad (\mathit{app})$$

app-elim

As expected, beta-reduction remains standard.

$$\frac{H : z \overset{A}{\mapsto} (\lambda w.e)y \;\longrightarrow\; (z \overset{A}{\mapsto} e[y/w])}{(H, z \overset{A}{\mapsto} (\lambda w.e)y) \overset{prod}{\Longrightarrow} (H, z \overset{A}{\mapsto} e[y/w])}$$

gives the following 1-processor case:

$$(H, z \overset{A}{\mapsto} (\lambda w.e)y) \underset{1}{\Longrightarrow} (H, z \overset{A}{\mapsto} e[y/w]) \qquad (\mathit{app\text{-}elim})$$

seq

A similar argument as used for the *app* rule leads to:

$$\frac{(H, z \overset{A}{\mapsto} e_1) \underset{1}{\Longrightarrow} (H', z \overset{\alpha}{\mapsto} e_1')}{(H, z \overset{A}{\mapsto} e_1 \;\mathbf{seq}\; e_2) \underset{1}{\Longrightarrow} (H', z \overset{\alpha}{\mapsto} e_1' \;\mathbf{seq}\; e_2)} \qquad (\mathit{seq})$$

seq-elim

This is similar to the *app-elim* rule too.

$$\frac{H : z \overset{A}{\mapsto} v \;\mathbf{seq}\; e \;\longrightarrow\; (z \overset{A}{\mapsto} e)}{(H, z \overset{A}{\mapsto} v \;\mathbf{seq}\; e) \overset{prod}{\Longrightarrow} (H, z \overset{A}{\mapsto} e)}$$

obviously gives:

$$(H, z \overset{A}{\mapsto} v \;\mathbf{seq}\; e) \underset{1}{\Longrightarrow} (H, z \overset{A}{\mapsto} e) \qquad (\mathit{seq\text{-}elim})$$

par

With a single processor, all uses of **par** become ineffective. For this reason (and the decision to directly activate in the $block_1$ rule) R-bindings can be removed. Bindings can only be activated from the main thread.

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

Changing the $\overset{R}{\mapsto}$ to $\overset{I}{\mapsto}$ thus leads to:

$$(H, z \overset{A}{\mapsto} x \textbf{ par } e_2) \underset{1}{\Longrightarrow} (H, z \overset{A}{\mapsto} e_2) \qquad (par)$$

par-elim

By inspection of the general rule, the x-binding is unchanged here too:

$$(H, x \overset{BAR}{\mapsto} e_1) : z \overset{A}{\mapsto} x \textbf{ par } e_2 \;\longrightarrow\; (z \overset{A}{\mapsto} e_2)$$

Hence, the derived 1-processor rule is the same as *par* above.

Single-processor semantic rules

The case by case derivation given above justifies our claim that our regular semantics restricted to a single processor is equivalent to the following set of rules.

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

$$(H, x \overset{I}{\mapsto} e, z \overset{A}{\mapsto} x) \underset{1}{\Longrightarrow} (H, z \overset{B}{\mapsto} x, x \overset{A}{\mapsto} e) \qquad (var)$$

$$(H, x \overset{I}{\mapsto} v, z \overset{A}{\mapsto} x) \underset{1}{\Longrightarrow} (H, x \overset{I}{\mapsto} v, z \overset{A}{\mapsto} \widehat{v}) \qquad (var\text{-}elim)$$

$$(H, x \overset{B}{\mapsto} e^{(z)}, z \overset{A}{\mapsto} v) \underset{1}{\Longrightarrow} (H, z \overset{I}{\mapsto} v, x \overset{A}{\mapsto} e^{(z)}) \qquad (val)$$

$$\frac{(H, z \overset{A}{\mapsto} e) \underset{1}{\Longrightarrow} (H', z \overset{\alpha}{\mapsto} e')}{(H, z \overset{A}{\mapsto} e\ y) \underset{1}{\Longrightarrow} (H', z \overset{\alpha}{\mapsto} e'\ y)} \qquad (app)$$

$$(H, z \overset{A}{\mapsto} (\lambda w.e) y) \underset{1}{\Longrightarrow} (H, z \overset{A}{\mapsto} e[y/w]) \qquad (app\text{-}elim)$$

$$\frac{(H, z \overset{A}{\mapsto} e_1) \underset{1}{\Longrightarrow} (H', z \overset{\alpha}{\mapsto} e_1')}{(H, z \overset{A}{\mapsto} e_1 \textbf{ seq } e_2) \underset{1}{\Longrightarrow} (H', z \overset{\alpha}{\mapsto} e_1' \textbf{ seq } e_2)} \qquad (seq)$$

$$(H, z \overset{A}{\mapsto} v \textbf{ seq } e) \underset{1}{\Longrightarrow} (H, z \overset{A}{\mapsto} e) \qquad (seq\text{-}elim)$$

$$(H, z \overset{A}{\mapsto} x \textbf{ par } e_2) \underset{1}{\Longrightarrow} (H, z \overset{A}{\mapsto} e_2) \qquad (par)$$

In the *seq* and *app* rules, α can be either A or B.

Relating the 1-processor and n-processor case

I need to prove that our semantics gives the same result for any positive number of processors. Or that it gives the same result as the 1-processor semantics, if that's easier.

Relating our semantics to the natural semantics.

This is the core of the proof that our semantics corresponds to John Launchbury's 'standard' natural semantics. First we need to extend his semantics to include **par** and **seq**.

Natural semantics

$$\Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e \qquad (\mathit{Lambda})$$

$$\frac{\Gamma : e \Downarrow \Delta : \lambda y.e' \qquad \Delta : e'[x/y] \Downarrow \Theta : v}{\Gamma : e\, x \Downarrow \Theta : v} \qquad (\mathit{Appln})$$

$$\frac{\Gamma : e \Downarrow \Delta : v}{(\Gamma, x \mapsto e) : x \Downarrow (\Delta, x \mapsto v) : \widehat{v}} \qquad (\mathit{Vble})$$

$$\frac{(\Gamma, x_1 \mapsto e_1, \ldots x_n \mapsto e_n) : e \Downarrow \Delta : v}{\Gamma : \mathbf{let}\ \{x_i = e_i\}_{i=1}^{n}\ \mathbf{in}\ e \Downarrow \Delta : v} \qquad (\mathit{Let})$$

$$\frac{\Gamma : e_1 \Downarrow \Delta : v_1 \qquad \Delta : e_2 \Downarrow \Theta : v_2}{\Gamma : e_1\ \mathbf{seq}\ e_2 \Downarrow \Theta : v_2} \qquad (\mathit{Seq})$$

$$\frac{\Gamma : e \Downarrow \Delta : v}{\Gamma : x\ \mathbf{par}\ e \Downarrow \Delta : v} \qquad (\mathit{Par})$$

The congruence result

We need a way of relating our heaps (bindings labelled with I, R, A, B) with Launchbury's heaps. We want to erase binding labels in one direction and add labels in the other direction. It is sufficient to label all bindings as being *inactive* since any other binding types reflect information about the computation that has gone before and have no influence on the evaluation of the current expression.

Definition
If H is a heap then $\overline{H}$ is a JL-heap with the same bindings but no binding labels.

If Δ is a JL-heap then Δ_I is a heap with the same bindings, all labelled *inactive*.

Both functions can be defined in the usual inductive way.

We wish to prove that any expression e reduces to a value v in our semantics if and only if e evaluates to v in Launchbury's semantics.

The following lemmas will be useful. First, adding inactive bindings to the heap does not affect the computation.

Lemma 3
If $H \underset{1}{\Longrightarrow} H'$ and x is not bound in H or H' then for any expression e and any $\alpha \neq A$ we have $(H, x \overset{\alpha}{\mapsto} e) \underset{1}{\Longrightarrow} (H', x \overset{\alpha}{\mapsto} e)$.

Proof By induction on the derivation of $H \underset{1}{\Longrightarrow} H'$.

In all rules except *app* and *seq* the heap is unchanged (and hence irrelevant to the rule) except for the distinguished bindings. Hence we may choose H arbitrarily.

The inductive cases are equally straightforward. For example, *app*:

$$\frac{(H, z \overset{A}{\mapsto} e) \underset{1}{\Longrightarrow} (H', z \overset{\alpha}{\mapsto} e')}{(H, z \overset{A}{\mapsto} e\, y) \underset{1}{\Longrightarrow} (H', z \overset{\alpha}{\mapsto} e'\, y)}$$

By induction we may assume

$$(H, z \overset{A}{\mapsto} e, x \overset{\beta}{\mapsto} e'') \underset{1}{\Longrightarrow} (H', z \overset{\alpha}{\mapsto} e', x \overset{\beta}{\mapsto} e'')$$

then by the *app* rule we have:

$$(H, z \overset{A}{\mapsto} e\, y, x \overset{\beta}{\mapsto} e'') \underset{1}{\Longrightarrow} (H', z \overset{\alpha}{\mapsto} e'\, y, x \overset{\beta}{\mapsto} e'')$$

□

As a kind of dual to lemma 3, there are circumstances where it is safe to remove certain bindings from the heap.

Lemma 4
If an expression reduces to whnf then it does not enter a black hole.
That is, if $(H, z \overset{B}{\mapsto} x, x \overset{A}{\mapsto} e) \underset{1}{\Longrightarrow}^* (H', z \overset{B}{\mapsto} x, x \overset{A}{\mapsto} v)$ then $(H, x \overset{A}{\mapsto} e) \underset{1}{\Longrightarrow}^* (H', x \overset{A}{\mapsto} v)$.

Proof By inspection of the rules, blocked bindings only come into play in the *val* rule. But then the active expression has already reached whnf so the computation sequence is the trivial one:

$$(H, z \overset{B}{\mapsto} x, x \overset{A}{\mapsto} v) \underset{1}{\Longrightarrow}^* (H', z \overset{B}{\mapsto} x, x \overset{A}{\mapsto} v)$$

and $(H, x \overset{A}{\mapsto} v) \underset{1}{\Longrightarrow}^* (H', x \overset{A}{\mapsto} v)$ is also trivially true. □

Another useful result is that the reduction of an application expression $e\, x$ proceeds by first reducing the operator e to whnf.

Lemma 5
$(H, z \overset{A}{\mapsto} e) \underset{1}{\Longrightarrow}^* (H', z \overset{A}{\mapsto} v)$ iff $(H, z \overset{A}{\mapsto} e\, x) \underset{1}{\Longrightarrow}^* (H', z \overset{A}{\mapsto} v\, x)$.
The two sequences are of the same length.

Proof For each individual step in $(H, z \overset{A}{\mapsto} e) \underset{1}{\Longrightarrow}^* (H', z \overset{A}{\mapsto} v)$ where z is active, e.g. $(H_i, z \overset{A}{\mapsto} e_i) \underset{1}{\Longrightarrow} (H_i', z \overset{\alpha}{\mapsto} e_i')$, the (*app*) rule gives $(H_i, z \overset{A}{\mapsto} e_i\, x) \underset{1}{\Longrightarrow} (H_i', z \overset{\alpha}{\mapsto} e_i'\, x)$.

If z is not active then it must be blocked, e.g. $(H_j, z \overset{B}{\mapsto} e_j) \underset{1}{\Longrightarrow} (H_j', z \overset{\beta}{\mapsto} e_j')$. In that case we can apply lemma 4 and then lemma 3 to show that $(H_j, z \overset{B}{\mapsto} e_j\, x) \underset{1}{\Longrightarrow} (H_j', z \overset{\beta}{\mapsto} e_j'\, x)$.

Stringing these steps together in corresponding order to the original computation gives the required sequence, clearly of the same length.

The reverse direction of the proof is similar. □

There is an equivalent result for **seq**-expressions:

Lemma 6
$(H, z \overset{A}{\mapsto} e_1) \underset{1}{\Longrightarrow}^* (H', z \overset{A}{\mapsto} v)$ iff $(H, z \overset{A}{\mapsto} e_1 \,\mathbf{seq}\, e_2) \underset{1}{\Longrightarrow}^* (H', z \overset{A}{\mapsto} v \,\mathbf{seq}\, e_2)$.
The two sequences are of the same length.

Proof Similar to lemma 5 using the (*seq*) rule. □

The two directions of the main result will be given as separate theorems.

Theorem 7
If $\Gamma : e \Downarrow \Delta : v$ then $(\Gamma_I, z \overset{A}{\mapsto} e) \underset{1}{\Longrightarrow}^* (\Delta_I, z \overset{A}{\mapsto} v)$ for some fresh variable z.

Proof The proof is by induction on the length of the natural semantics derivation.

- (*Lambda*)

$$\Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e$$

  By the reflexivity of $\underset{1}{\Longrightarrow}^*$, we immediately have:

$$(\Gamma_I, z \overset{A}{\mapsto} \lambda x.e) \underset{1}{\Longrightarrow}^* (\Gamma_I, z \overset{A}{\mapsto} \lambda x.e)$$

- (*Appln*)

$$\frac{\Gamma : e \Downarrow \Delta : \lambda y.e' \qquad \Delta : e'[x/y] \Downarrow \Theta : v}{\Gamma : e\, x \Downarrow \Theta : v}$$

  By induction we may assume:

$$(\Gamma_I, z \overset{A}{\mapsto} e) \underset{1}{\Longrightarrow}^* (\Delta_I, z \overset{A}{\mapsto} \lambda y.e') \tag{1}$$

$$(\Delta_I, z \overset{A}{\mapsto} e'[x/y]) \underset{1}{\Longrightarrow}^* (\Theta_I, z \overset{A}{\mapsto} v) \tag{2}$$

  Applying lemma 5 to (1) gives:

$$(\Gamma_I, z \overset{A}{\mapsto} e\, x) \underset{1}{\Longrightarrow} (\Delta_I, z \overset{A}{\mapsto} (\lambda y.e')x)$$

  By (*app-elim*) and (2) we have

$$\begin{aligned} &\underset{1}{\Longrightarrow} (\Delta_I, z \overset{A}{\mapsto} e'[x/y]) \\ &\underset{1}{\Longrightarrow}^* (\Theta_I, z \overset{A}{\mapsto} v) \end{aligned}$$

- (*Vble*)

$$\frac{\Gamma : e \Downarrow \Delta : v}{(\Gamma, x \mapsto e) : x \Downarrow (\Delta, x \mapsto v) : \widehat{v}}$$

  By induction we may assume:

$$(\Gamma_I, x \overset{A}{\mapsto} e) \underset{1}{\Longrightarrow}^* (\Delta_I, x \overset{A}{\mapsto} v) \tag{3}$$

  So by lemma 3:

$$(\Gamma_I, z \overset{B}{\mapsto} x, x \overset{A}{\mapsto} e) \underset{1}{\Longrightarrow}^* (\Delta_I, z \overset{B}{\mapsto} x, x \overset{A}{\mapsto} v) \tag{4}$$

Hence we can construct the following computation:

$$
\begin{aligned}
(\Gamma_I, x \overset{I}{\mapsto} e, z \overset{A}{\mapsto} x) &\underset{1}{\Longrightarrow} (\Delta_I, z \overset{B}{\mapsto} x, x \overset{A}{\mapsto} e) && \text{(by } \mathit{var}\text{)}\\
&\underset{1}{\Longrightarrow}^* (\Delta_I, z \overset{B}{\mapsto} x, x \overset{A}{\mapsto} v) && \text{(by 4)}\\
&\underset{1}{\Longrightarrow} (\Delta_I, x \overset{I}{\mapsto} v, z \overset{A}{\mapsto} x) && \text{(by } \mathit{val}\text{)}\\
&\underset{1}{\Longrightarrow} (\Delta_I, x \overset{I}{\mapsto} v, z \overset{A}{\mapsto} \widehat{v}) && \text{(by } \mathit{var\text{-}elim}\text{)}
\end{aligned}
$$

as required.

- (*Let*)

$$\frac{(\Gamma, \{x_i \mapsto e_i\}_{i=1}^n) : e \Downarrow \Delta : v}{\Gamma : \textbf{let } \{x_i = e_i\}_{i=1}^n \textbf{ in } e \Downarrow \Delta : v}$$

We have the following reduction:

$$
\begin{aligned}
(\Gamma_I, z \overset{A}{\mapsto} \textbf{let } \{x_i = e_i\}_{i=1}^n \textbf{ in } e) &\underset{1}{\Longrightarrow} (\Gamma_I, \{x_i \overset{I}{\mapsto} e_i\}_{i=1}^n, z \overset{A}{\mapsto} e) && \text{(by } \mathit{let}\text{)}\\
&\underset{1}{\Longrightarrow} (\Delta_I, z \overset{A}{\mapsto} v) && \text{(by induction)}
\end{aligned}
$$

- (*Seq*)

$$\frac{\Gamma : e_1 \Downarrow \Delta : v_1 \qquad \Delta : e_2 \Downarrow \Theta : v_2}{\Gamma : e_1 \textbf{ seq } e_2 \Downarrow \Theta : v_2}$$

Similar to the *Appln* case. By induction we may assume:

$$(\Gamma_I, z \overset{A}{\mapsto} e_1) \underset{1}{\Longrightarrow}^* (\Delta_I, z \overset{A}{\mapsto} v_1) \tag{5}$$

$$(\Delta_I, z \overset{A}{\mapsto} e_2) \underset{1}{\Longrightarrow}^* (\Theta_I, z \overset{A}{\mapsto} v_2) \tag{6}$$

Applying lemma 6 to (5) gives:

$$(\Gamma_I, z \overset{A}{\mapsto} e_1 \textbf{ seq } e_2) \underset{1}{\Longrightarrow}^* (\Delta_I, z \overset{A}{\mapsto} v_1 \textbf{ seq } e_2)$$

By (*seq-elim*) and (6) we have

$$
\begin{aligned}
&\underset{1}{\Longrightarrow} (\Delta_I, z \overset{A}{\mapsto} e_2)\\
&\underset{1}{\Longrightarrow}^* (\Theta_I, z \overset{A}{\mapsto} v_2)
\end{aligned}
$$

- (*Par*)

$$\frac{\Gamma : e \Downarrow \Delta : v}{\Gamma : x \textbf{ par } e \Downarrow \Delta : v}$$

Trivial. By induction, $(\Gamma_I, z \overset{A}{\mapsto} e) \underset{1}{\Longrightarrow}^* (\Delta_I, z \overset{A}{\mapsto} v)$. Rule *par* immediately gives the required result.

□

Theorem 8
If $(H, z \overset{A}{\mapsto} e) \underset{1}{\Longrightarrow}^* (H', z \overset{A}{\mapsto} v)$ then $\overline{H} : e \Downarrow \overline{H}' : v$ for some fresh variable z.

Proof The proof is by induction on the length of the reduction $\underset{1}{\Longrightarrow}^*$.

- (*let*)

$$\begin{aligned}(H, z \overset{A}{\mapsto} \textbf{let}\ \{x_i = e_i\}_{i=1}^n\ \textbf{in}\ e) &\underset{1}{\Longrightarrow} (H, \{x_i \overset{I}{\mapsto} e_i\}_{i=1}^n, z \overset{A}{\mapsto} e) \\ &\underset{1}{\Longrightarrow}^* (H', z \overset{A}{\mapsto} v)\end{aligned}$$

By induction, $(\Gamma, \{x_i \mapsto e_i\}_{i=1}^n) : e \Downarrow \Delta : v$.
By *Let* we have $\quad \Gamma : \textbf{let}\ \{x_i = e_i\}_{i=1}^n\ \textbf{in}\ e \Downarrow \Delta : v \quad$ as required.

- (*var*)

$$\begin{aligned}(H, x \overset{I}{\mapsto} e, z \overset{A}{\mapsto} x) &\underset{1}{\Longrightarrow} (H, z \overset{B}{\mapsto} x, x \overset{A}{\mapsto} e) \\ &\underset{1}{\Longrightarrow} (H', z \overset{B}{\mapsto} x, x \overset{A}{\mapsto} v) && (1) \\ &\underset{1}{\Longrightarrow} (H, x \overset{I}{\mapsto} v, z \overset{A}{\mapsto} \widehat{v}) && \text{(by } \textit{val}\text{)}\end{aligned}$$

Applying lemma 4 to (1) gives:

$$(H, x \overset{A}{\mapsto} e) \underset{1}{\Longrightarrow}^* H', x \overset{A}{\mapsto} v$$

Hence by induction we have $\overline{H} : e \Downarrow \overline{H}' : v$ and by rule *Vble* we have $(\overline{H}, x \mapsto e) : x \Downarrow (\overline{H}', x \mapsto v) : \widehat{v}$ as required.

- (*var-elim*)

$$(H, x \overset{I}{\mapsto} v, z \overset{A}{\mapsto} x) \underset{1}{\Longrightarrow} (H, x \overset{I}{\mapsto} v, z \overset{A}{\mapsto} \widehat{v})$$

Values are lambda expressions so setting v to $\lambda y.e$, we can construct the derivation:

$$\frac{\overline{H} : \lambda y.e \Downarrow \overline{H} : \lambda y.e}{(\overline{H}, x \mapsto \lambda y.e) : x \Downarrow (\overline{H}, x \mapsto \lambda y.e) : \widehat{\lambda y.e}}$$

as required.

- (*val*)

$$(H, x \overset{B}{\mapsto} e^{(z)}, z \overset{A}{\mapsto} v) \underset{1}{\Longrightarrow} (H, z \overset{I}{\mapsto} v, x \overset{A}{\mapsto} e^{(z)})$$

This step can never begin a sequence that leads to a configuration with z active. The result follows trivially.

- (*app*)

$$\begin{aligned}(H, z \overset{A}{\mapsto} e\ y) &\underset{1}{\Longrightarrow}^* (H', z \overset{A}{\mapsto} (\lambda w.e')y) && (2) \\ &\underset{1}{\Longrightarrow} (H', z \overset{A}{\mapsto} e'[y/w]) && \text{(by } \textit{app-elim}\text{)} \\ &\underset{1}{\Longrightarrow}^* (H'', z \overset{A}{\mapsto} v) && (3)\end{aligned}$$

Applying lemma 5 to (2) gives:

$$(H, z \overset{A}{\mapsto} e) \underset{1}{\Longrightarrow}^* (H', z \overset{A}{\mapsto} \lambda w.e') \qquad (4)$$

By induction on (4) and (3) we may therefore assume:

$$\begin{aligned} \overline{H} : e \Downarrow \overline{H}' : \lambda w.e' \\ \overline{H}' : e'[y/w] \Downarrow \overline{H}'' : v \end{aligned}$$

Now by the *Appln* rule we have $\overline{H} : e\, y \Downarrow \overline{H}'' : v$.

- (*app-elim*)

$$\begin{aligned} (H, z \overset{A}{\mapsto} (\lambda w.e')y) &\underset{1}{\Longrightarrow} (H, z \overset{A}{\mapsto} e'[y/w]) && \text{(by } \textit{app-elim}\text{)} \\ &\underset{1}{\Longrightarrow}^{*} (H', z \overset{A}{\mapsto} v) && (5) \end{aligned}$$

Use *Lambda* to give the first premise of the *Appln* rule and the inductive hypothesis on (5) to give the second:

$$\frac{\overline{H} : \lambda w.e \Downarrow \overline{H} : \lambda w.e \qquad \overline{H} : e[y/w] \Downarrow \overline{H}' : v}{\overline{H} : (\lambda w.e)y \Downarrow \overline{H}' : v}$$

- (*seq*)
This closely follows the proof for *app*, using lemma 6 in place of lemma 5. (Exercise!)

- (*seq-elim*)

$$\begin{aligned} (H, z \overset{A}{\mapsto} v\ \mathbf{seq}\ e) &\underset{1}{\Longrightarrow} (H, z \overset{A}{\mapsto} e) \\ &\underset{1}{\Longrightarrow}^{*} (H', z \overset{A}{\mapsto} v) && (6) \end{aligned}$$

Similar to the *app-elim* case, use *Lambda* to give the first premise of the *Seq* rule and the inductive hypothesis on (6) to give the second:

$$\frac{\overline{H} : v \Downarrow \overline{H} : v \qquad \overline{H} : e \Downarrow \overline{H}' : v'}{\overline{H} : v\ \mathbf{seq}\ e \Downarrow \overline{H}' : v'}$$

- (*par*)

$$\begin{aligned} (H, z \overset{A}{\mapsto} x\ \mathbf{par}\ e) &\underset{1}{\Longrightarrow} (H, z \overset{A}{\mapsto} e) \\ &\underset{1}{\Longrightarrow} (H', z \overset{A}{\mapsto} v) && (7) \end{aligned}$$

Using the inductive hypothesis on (7) to give the premise of the *Par* rule:

$$\frac{\overline{H} : e \Downarrow \overline{H}' : v}{\overline{H} : x\ \mathbf{par}\ e \Downarrow \overline{H}' : v}$$

□

Correctness of the natural semantics.

In John Launchbury's paper, he demonstrated that his natural semantics was correct with respect to a standard denotational semantics. Since we have extended the language with **seq** and **par** we should also extend his proof. Fortunately, his proof is given in detail and is very clear.

This section is not intended to be self-contained and should be read in conjunction with JL's paper. We do not repeat definitions and minor results. As far as possible, we borrow his notation.

First we define the denotational semantics of **seq** and **par**:

$$\llbracket e_1 \textbf{ seq } e_2 \rrbracket_\rho = \begin{cases} \bot & \text{if } \llbracket e_1 \rrbracket_\rho = \bot \\ \llbracket e_2 \rrbracket_\rho & \text{otherwise} \end{cases}$$

$$\llbracket x \textbf{ par } e \rrbracket_\rho = \llbracket e \rrbracket_\rho$$

Correctness

Launchbury's first substantial result is the following:

Theorem 9 (JL Theorem 2)
If $\Gamma : e \Downarrow \Delta : v$ then for all environments ρ,

$$\llbracket e \rrbracket_{\{\!\{\Gamma\}\!\}\rho} = \llbracket v \rrbracket_{\{\!\{\Delta\}\!\}\rho} \quad \wedge \quad \{\!\{\Gamma\}\!\}\rho \leq \{\!\{\Delta\}\!\}\rho$$

Proof The proof is by induction on the structure of the derivation of $\Gamma : e \Downarrow \Delta : v$. Here are the cases for the language extensions:

- *Seq*

$$\frac{\Gamma : e_1 \Downarrow \Delta : v_1 \qquad \Delta : e_2 \Downarrow \Theta : v_2}{\Gamma : e_1 \textbf{ seq } e_2 \Downarrow \Theta : v_2}$$

  The second part is easy. By induction we have $\{\!\{\Gamma\}\!\}\rho \leq \{\!\{\Delta\}\!\}\rho$ and $\{\!\{\Delta\}\!\}\rho \leq \{\!\{\Theta\}\!\}\rho$ so by transitivity of $\leq$ we have $\{\!\{\Gamma\}\!\}\rho \leq \{\!\{\Theta\}\!\}\rho$ as required.

  Now applying the following lemma 10 to this result we have:

$$\llbracket e_2 \rrbracket_{\{\!\{\Gamma\}\!\}\rho} = \llbracket e_2 \rrbracket_{\{\!\{\Delta\}\!\}\rho}$$

  and by induction:

$$\llbracket e_2 \rrbracket_{\{\!\{\Delta\}\!\}\rho} = \llbracket v_2 \rrbracket_{\{\!\{\Theta\}\!\}\rho}$$

  So $\llbracket e_1 \textbf{ seq } e_2 \rrbracket_{\{\!\{\Delta\}\!\}\rho} = \llbracket v_2 \rrbracket_{\{\!\{\Theta\}\!\}\rho}$.

- *Par* As would be expected, this case is trivial.

□

It seems to me that Launchbury assumes the following lemma. For example, see the change of environment of x in the $(e\ x)$ case of his theorem 2.

Lemma 10
If $\llbracket e \rrbracket_\rho = v$ and $\rho \leq \rho'$ then $\llbracket e \rrbracket_{\rho'} = v$

Proof An easy structural induction on e. □

Computational adequacy

To begin, we extend the resourced semantics to include **seq** and **par**:

$$\mathcal{N}[\![e_1 \;\mathbf{seq}\; e_2]\!]_\sigma(Sk) = \begin{cases} \bot & \text{if } \mathcal{N}[\![e_1]\!]_\sigma k = \bot \\ \mathcal{N}[\![e_2]\!]_\sigma k & \text{otherwise} \end{cases}$$
$$\mathcal{N}[\![x \;\mathbf{par}\; e]\!]_\sigma = \mathcal{N}[\![e]\!]_\sigma$$

In Launchbury's 'alternative version' of the natural semantics, there is no need to change the rules for **seq** or **par**.

Lemma 11 (JL Lemma 5)
If $[\![e]\!]_\rho \neq \bot$ then there exists a natural number m such that $\mathcal{N}[\![e]\!]_\sigma(S^m\bot) \neq \bot$ where $\forall x \,.\, \rho\, x = \sigma\, x\, \omega$.

Proof Since the **seq** and **par** cases of the definition of $\mathcal{N}$ are also in terms of continuous functions, then if the original semantics assigns a non-bottom value to a term, so does some finite approximation (paraphrasing Launchbury). □

Lemma 12 (JL Lemma 6)
If $\mathcal{N}[\![e]\!]_{\mu\sigma.(x_1 \mapsto \mathcal{N}[\![e_1]\!]_\sigma \sqcup \ldots x_n \mapsto \mathcal{N}[\![e_n]\!]_\sigma)} S^m\bot \neq \bot$, then there exists a heap Δ and a value v such that $(x_1 \mapsto e_1 \ldots x_n \mapsto e_n) : e \Downarrow \Delta : v$ in the alternative version of the semantics.

Proof By induction on m. The case for **seq** follows. The **par** case is trivial.

Suppose $\mathcal{N}[\![e \;\mathbf{seq}\; e']\!]_{\mu\sigma.(x_1 \mapsto \mathcal{N}[\![e_1]\!]_\sigma \sqcup \ldots x_n \mapsto \mathcal{N}[\![e_n]\!]_\sigma)} S^m\bot \neq \bot$

Then by definition of $\mathcal{N}$:

$$\mathcal{N}[\![e]\!]_{\mu\sigma.(x_1 \mapsto \mathcal{N}[\![e_1]\!]_\sigma \sqcup \ldots x_n \mapsto \mathcal{N}[\![e_n]\!]_\sigma)} S^{m-1}\bot \neq \bot$$
$$\mathcal{N}[\![e']\!]_{\mu\sigma.(x_1 \mapsto \mathcal{N}[\![e_1]\!]_\sigma \sqcup \ldots x_n \mapsto \mathcal{N}[\![e_n]\!]_\sigma)} S^{m-1}\bot \neq \bot$$

Hence by induction:

$$(x_1 \mapsto e_1 \ldots x_n \mapsto e_n) : e \Downarrow \Delta : v \qquad \text{some } \Delta, v \tag{1}$$
$$(x_1 \mapsto e_1 \ldots x_n \mapsto e_n) : e' \Downarrow \Theta : v' \qquad \text{some } \Theta, v' \tag{2}$$

Applying theorem 9 to (1) gives:

$$\{\!\{(x_1 \mapsto e_1 \ldots x_n \mapsto e_n)\}\!\}\rho_0 \leq \{\!\{\Delta\}\!\}\rho_0$$

so by lemma 10 and (2):

$$\Delta : e' \Downarrow \Theta : v' \tag{3}$$

Using (1) and (3) as premises to the *Seq* rule gives the required result:

$$(x_1 \mapsto e_1 \ldots x_n \mapsto e_n) : e \;\mathbf{seq}\; e' \Downarrow \Theta : v'$$

□

Launchbury's final result for the extended language follows by his proof without modification:

Theorem 13 (JL Theorem 7)
$[\![e]\!]_{\{\!\{\Gamma\}\!\}\rho_0} \neq \bot \quad \Rightarrow \quad \exists \Delta, v \,.\, \Gamma : e \Downarrow \Delta : v$ □