JFP: page 1 of 40. © Cambridge University Press 2012
doi:10.1017/S0956796812000214

The Reduceron reconfigured and re-evaluated

MATTHEW NAYLOR and COLIN RUNCIMAN
Department of Computer Science, University of York, York, North Yorkshire, UK
(*e-mail:* {mfn,colin}@cs.york.ac.uk)

Abstract

A new version of a special-purpose processor for running lazy functional programs is presented. This processor – the Reduceron – exploits parallel memories and dynamic analyses to increase evaluation speed, and is implemented using reconfigurable hardware. Compared to a more conventional functional language implementation targeting a standard RISC processor running on the same reconfigurable hardware, the Reduceron offers a significant improvement in run-time performance.

1 Introduction

Efficient evaluation of high-level functional programs on conventional computers is a big challenge. Sophisticated techniques are needed to exploit architectural features designed for low-level imperative execution. Furthermore, conventional computers have limitations when it comes to running functional programs. For example, memory bandwidth is limited to *serial* communication in *small units*. Evaluators based on graph reduction perform intensive construction and deconstruction of expressions in memory. Each such operation requires sequential execution of *many* machine instructions, not because of any inherent data dependencies, but because of architectural constraints in conventional computers.

All this motivates the idea of computers specially designed to meet the needs of high-level functional languages – much as GPUs are designed to meet needs in graphics. By providing a minimal set of features tailored to the execution of functional programs, such a *custom computer* could be *not only fast but simple*, with benefits such as fuller verification and lower energy consumption. This is not a new idea. In the 1980s and 1990s there was a 15-year ACM conference series *Functional Programming Languages and Computer Architecture*. In separate initiatives, there was an entire workshop concerned with graph-reduction machines alone (Fasel & Keller, 1987), and a major computer manufacturer built a graph-reduction prototype (Scheevel, 1986). But the process of constructing exotic new hardware was slow and uncertain. With major advances in compilation for ever bigger, faster and cheaper mass-market machines, the idea of specialised hardware for functional languages went out of fashion.

Reconfigurable hardware. Today, the situation is quite different. *Field-programmable gate arrays* (FPGAs) have greatly reduced the effort and expertise needed to develop special-purpose hardware. They contain thousands of parallel logic blocks that can be configured at will by software tools. They are widely available and represent an advancing technology in its own right.

The downside of FPGA applications is that they typically have much lower maximum clocking frequencies than corresponding directly fabricated circuits – this is the price we pay for *configurability*. To obtain good performance using an FPGA, it is therefore necessary to exploit significant parallelism.

The Reduceron. In this paper, we present a special-purpose machine for sequential graph reduction – the Reduceron – implemented on an FPGA. We build upon our previous work on the same topic (Naylor & Runciman, 2008) by presenting a new design that exhibits a factor of five performance improvement.

A notable feature of our new design is that each of its six semantic reduction rules is performed in a *a single clock-cycle*. All the necessary memory transactions required to perform a reduction are done *in parallel*. The Reduceron performs on average 0.55 *hand-reductions* per clock-cycle. A hand-reduction is a reduction that a programmer would perform in *by-hand* evaluation trace of a program; it includes function application and case analysis, but not machine-level reductions such as updating and unwinding.

Another notable development in our new design is the use of two *dynamic analyses* enabling *update avoidance* and *speculative evaluation of primitive redexes*, both of which lead to significant performance improvements. On conventional computers, the run-time overhead of these dynamic analyses would be prohibitive, but on an FPGA they are cheap and simple to implement.

Contributions. In summary, we give the following:

Section 2. A precise description of the Reduceron compiler, including refinements to the Scott encoding of constructors, used for compiling case expressions and addressing various efficiency concerns.
Section 3. An operational semantics of the template instantiation machine underpinning the Reduceron implementation.
Section 4. A detailed description of how each semantic reduction rule is implemented in a single clock-cycle using an FPGA.
Section 5. Extensions to the semantics to support (1) dynamic sharing analysis, used to avoid unnecessary heap updates, and (2) dynamic detection of primitive redexes, enabling speculative reduction of such expressions during function-body instantiation.
Section 6. A comparative evaluation of the Reduceron implementation against other functional language implementations.

Extensions. This paper is an extended version of a conference paper (Naylor & Runciman, 2010). The main new developments are as follows:

- Figure 2 added to formalise the Reduceron compilation scheme.
- Section 4.3 extended with more details of our template-splitting algorithm.

$$
\begin{array}{lcll}
e & ::= & \vec{e} & \text{(Application)} \\
 & | & \texttt{case}\ e\ \texttt{of}\ \{\,\vec{a}\,\} & \text{(Case Expression)} \\
 & | & \texttt{let}\ \{\,\vec{b}\,\}\ \texttt{in}\ e & \text{(Let Expression)} \\
 & | & n & \text{(Integer)} \\
 & | & x & \text{(Variable)} \\
 & | & p & \text{(Primitive)} \\
 & | & f & \text{(Function)} \\
 & | & C & \text{(Constructor)} \\
 & | & \texttt{<}\vec{f}\texttt{>} & \text{(Case Table)} \\
a & ::= & C\,\vec{x}\ \texttt{->}\ e & \text{(Case Alternative)} \\
b & ::= & x\ \texttt{=}\ e & \text{(Let Binding)} \\
d & ::= & f\,\vec{x}\ \texttt{=}\ e & \text{(Function Definition)}
\end{array}
$$

Fig. 1. Core syntax of F-lite.

- Section 6.1 extended to include a brief description of each benchmark program.
- Section 6.3 extended with a comparison of the Reduceron against a conventionally compiled functional language implementation targeting a Reduced Instruction Set Computer (RISC) processor running on the same FPGA. (This is the most significant new contribution, reflected by the addition of the word 're-evaluated' to the paper title).
- Section 6.9 extended with more details of our initial experiments with static primitive-redex speculation (PRS).
- Sections 6.10, 6.12, 6.13 and 6.14 added to broaden discussion of related work.

2 Compilation

This section defines a series of refinements that take programs written in a lazy functional language called *F-lite* to a form known as *template code*, which the Reduceron can execute.

2.1 Source language

F-lite is a core lazy functional language, close to subsets of both Haskell and Clean. The syntax of F-lite is presented in Figure 1.

Case expressions. Case expressions are in a simplified form that can be produced by a pattern match compiler such as that defined in Peyton Jones (1987). Patterns in case alternatives are constructors applied to zero or more variables. All case expressions contain an alternative for every constructor of the case subject's type.

Primitives. The meta-variable p denotes a primitive function symbol. All applications of primitive functions are fully saturated. The Reduceron implements only a small set of primitive operations, not the full set of a conventional processor, e.g. we have no floating-point operations. Primitives used in this paper include `(+)`, `(-)` and `(<=)`.

Main. Every program contains a definition `main = ` e, where e is an expression that evaluates to an integer n; the result of the program is the value n.

Case tables. Note the unusual case-table construct $<\vec{f}>$. Case tables are not written in source programs but are introduced during compilation – see Section 2.4.

Examples. Here are two example function definitions. The first concatenates two lists and the second computes triangular numbers.

```
append xs ys = case xs of
  { Nil -> ys ; Cons x xs -> Cons x (append xs ys) }

tri n = case (<=) n 1 of
  { False -> (+) (tri ((-) n 1)) n ; True -> 1 }
```

2.2 Terminology

Application length. The *length* of an application $e_1 \cdots e_n$ is n. For example, the length of the application `append xs ys` is three.

Compound and atomic expressions. Applications, case expressions and let expressions are *compound* expressions. All other expressions are *atomic*.

Flat expression. A *flat* expression is an atomic expression or an application $e_1 \cdots e_n$ in which each e_i for i in $1 \cdots n$ is an atomic expression. For example, `append xs ys` is a flat expression, but `tri ((-) n 1)` is not.

Expression graph. A let expression

$$\texttt{let \{ } x_1 = e_1 \texttt{ ; } \cdots \texttt{ ; } x_n = e_n \texttt{ \} in } e$$

is an *expression graph* exactly if e is a flat expression and each e_i for i in $1 \cdots n$ is a flat expression. Expression graphs are restricted A-normal forms (Flanagan *et al.*, 1993).

Constructor index and arity. Each constructor C of a data type with m constructors is associated with a unique *index* in the range $1 \cdots m$. More precisely, the index of a constructor is its position in the *alphabetically sorted* list of all constructors of that data type. For example, the standard list data type has two constructors: `Cons` has index 1 and `Nil` has index 2. A constructor with index i is denoted C_i, and the arity of a constructor C is denoted $\#C$.

2.3 Primitive applications

In a lazy language, an application of a primitive function such as `(+)`, `(-)` or `(<=)` requires special treatment: The integer arguments must be fully evaluated before the application can be reduced. One simple approach is to transform binary primitive applications by the rule

$$p\ e_0\ e_1 \quad \rightarrow \quad e_1\ (e_0\ p) \tag{1}$$

with the run-time reduction rule

$$n\ e \quad \rightarrow \quad e\ n \tag{2}$$

for any fully evaluated integer literal *n*. To illustrate this approach, consider the expression `(+) (tri 1) (tri 2)`. By compile-time application of rule (1), the expression is transformed to `tri 2 ((tri 1) (+))`. At run-time, reduction is as follows.

$$\begin{array}{lll} & \texttt{tri 2 ((tri 1) (+))} & \{\ \texttt{tri 2}\ \text{evaluates to}\ \texttt{3}\ \} \\ = & \texttt{3 ((tri 1) (+))} & \{\ \text{Rule (2)}\ \} \\ = & \texttt{(tri 1) (+) 3} & \{\ \texttt{tri 1}\ \text{evaluates to}\ \texttt{1}\ \} \\ = & \texttt{1 (+) 3} & \{\ \text{Rule (2)}\ \} \\ = & \texttt{(+) 1 3} & \end{array}$$

After transformation by rule (1), `tri` looks as follows.

```
tri n = case 1 (n (<=)) of
  { False -> n (tri (1 (n (-))) (+)) ; True -> 1 }
```

In Section 5, we present more efficient techniques for dealing with primitive applications.

2.4 Case expressions

This section describes how case expressions are compiled. First we recall the Scott encoding (Scott, 1968) recently rediscovered by Jansen *et al.* (2007). Then we make a number of refinements to this encoding.

The Scott/Jansen encoding. The first step of the encoding is to generate, for each constructor C_i of a data type with *m* constructors, a function definition

$$C_i\ x_1 \cdots x_{\#C_i}\ k_1 \cdots k_m \quad = \quad k_i\ x_1 \cdots x_{\#C_i} \tag{3}$$

The idea is that each data constructor C_i is encoded as a function that takes as arguments the $\#C_i$ arguments of the constructor and *m* continuations.The function encoding constructor C_i passes the constructor arguments to the *i*th continuation. For example, the list constructors are transformed to the following functions.

```
Cons x xs c n = c x xs
Nil       c n = n
```

Now case expressions of the form

$$\texttt{case}\ e\ \texttt{of \{}\ C_1\ \vec{x}_1\ \texttt{->}\ e_1\ \texttt{;} \cdots \texttt{;}\ C_m\ \vec{x}_m\ \texttt{->}\ e_m\ \texttt{\}} \tag{4}$$

are transformed to

$$e\ (alt_1\ \vec{v}_1\ \vec{x}_1) \cdots (alt_m\ \vec{v}_m\ \vec{x}_m)$$

where $\vec{v}_i$ are the free variables occurring in the *ith* case alternative and each alt_i for *i* in $1 \cdots m$ has the definition

$$alt_i\ \vec{v}_i\ \vec{x}_i \quad = \quad e_i$$

For example, the append function is transformed to

```
append xs ys = xs (consCase ys) (nilCase ys)
consCase ys x xs = Cons x (append xs ys)
nilCase  ys      = ys
```

Note that the application of `nilCase` could be reduced at compile-time. This is a consequence of constructor `Nil` having arity 0.

Larger example. Now let us look at a slightly larger example: An evaluator for basic arithmetic expressions.

```
eval x y e = case e of {
  Add n m -> (+) (eval x y n) (eval x y m);
  Neg n   -> (-) 0 (eval x y n);
  Sub n m -> (-) (eval x y n) (eval x y m);
  X       -> x;
  Y       -> y;
}
```

After transformation, and in-lining the nullary cases, we have

```
eval x y e = e (add x y) (neg x y) (sub x y) x y
add  x y n m = (+) (eval x y n) (eval x y m)
neg  x y n   = (-) 0 (eval x y n)
sub  x y n m = (-) (eval x y n) (eval x y m)
```

Look at the large body of `eval`: It contains three nested function applications and several repeated references to `x` and `y`. In typical functional language implementations, large function bodies are more expensive to construct than small ones.

Refinement 1. Rather than partially applying each case-alternative function to the free variables it refers to, we can define every alternative function alike to take *all* free variables occurring in *any* alternative. A case alternative can simply ignore variables that it does not need. So let us instead transform case expressions to

$$e\ alt_1 \cdots\ alt_m\ \vec{v}$$

where $\vec{v}$ is the union of the free variables in each case alternative, and each alt_i for i in $1 \cdots m$ has the definition

$$alt_i\ \vec{x}_i\ \vec{v}\ \ =\ \ e_i$$

Each case-alternative function now takes the constructor arguments followed by the free variables, rather than the other way around. To illustrate, append now looks as follows.

```
append xs ys = xs consCase nilCase ys
consCase x xs ys = Cons x (append xs ys)
nilCase       ys = ys
```

And `eval` becomes

```
eval x y e = e add neg sub xCase yCase x y
add   n m x y = (+) (eval x y n) (eval x y m)
neg   n   x y = (-) 0 (eval x y n)
sub   n m x y = (-) (eval x y n) (eval x y m)
xCase     x y = x
yCase     x y = y
```

The new bodies of `append` and `eval` contain *no* nested function applications and *no* repeated references. An apparent disadvantage is that we have had to introduce functions for the 0-arity constructor cases `nilCase`, `xCase` and `yCase`. But our next refinement prepares the way to recover the cost of applying these functions.

Refinement 2. We now have a large row of contiguous constants in the body of `eval`. To allow these constants to be represented efficiently (see Section 2.7) we place them in a *case table*. Case expressions are transformed to

$$e \; {<}alt_1 , \cdots , \; alt_m{>} \; \vec{v}$$

and each constructor C_i is encoded as

$$C_i \; x_1 \cdots x_{\#C_i} \; t \;\; = \;\; (t \; ! \; i) \;\; x_1 \cdots x_{\#C_i}$$

where $t!i$ returns the ith element of case table t.

Refinement 3. An evaluator can handle constructors more efficiently than general function definitions. We could introduce the following reduction rule for constructors.

$$C_i \; e_1 \cdots e_{\#C_i} \; t \quad \rightarrow \quad (t \; ! \; i) \;\; e_1 \cdots e_{\#C_i}$$

This rule replaces a constructor with a case-alternative function by looking up the case table using the constructor's index. However, the rule *also* drops the t argument. As a result, an implementation would have to slide the constructor arguments down the stack. A reduction rule that does not require argument sliding is

$$C_i \; e_1 \cdots e_{\#C_i} \; t \quad \rightarrow \quad (t \; ! \; i) \;\; e_1 \cdots e_{\#C_i} \; t \qquad (5)$$

To account for the fact that t has *not* been dropped, the case-alternative functions take the form

$$alt_i \; \vec{x}_i \; t \; \vec{v} \;\; = \;\; e_i$$

The final version of `append` is

```
append xs ys = xs <consCase, nilCase> ys
consCase x xs t ys = Cons x (append xs ys)
nilCase       t ys = ys
```

The `t` argument is simply ignored by the case alternatives. The final version of `tri` is

```
tri n = 1 (n (<=)) <falseCase, trueCase> n
falseCase t n = n (tri (1 (n (-))) (+))
trueCase  t n = 1
```

In Sections 3.3 and 4.7 we will see how these refinements enable efficient choices to be made at the implementation level.

2.5 In-lining

Our definition of append is no longer directly recursive. This is a consequence of splitting the case alternatives off as new function definitions. However, direct recursion is easily recovered: simply in-line the definition of append in the body of consCase.

```
consCase x xs t ys =
  Cons x (xs <consCase, nilCase> ys)
```

This transformation motivates the following general in-lining rule: *in-line saturated applications of functions that have flat bodies.* In-lining a flat expression e is often a big win because it eliminates a reduction and e is often no larger than the application it replaces.

2.6 Expression graphs

It is convenient for implementation purposes to make the graph structure of function bodies explicit by transforming them to expression graphs (Section 2.2). This is achieved by three rewrite rules.
(1) Lift nested applications into let bindings:

$$e_1 \cdots \; (e_i) \cdots \; e_n \;\; \rightarrow \;\; \texttt{let} \; \{ \; x \; = \; e_i \; \} \; \texttt{in} \; e_1 \cdots x \cdots e_n$$

where e_i is an application or a let expression, and x is a fresh variable.
(2) Lift let expressions out of let bodies:

$$\texttt{let} \; \{ \; \vec{b}_0 \; \} \; \texttt{in} \; (\texttt{let} \; \{ \; \vec{b}_1 \; \} \; \texttt{in} \; e) \rightarrow \texttt{let} \; \{ \; \vec{b}_0 \; ; \; \vec{b}_1 \; \} \; \texttt{in} \; e$$

(3) Lift let expressions out of let bindings:

$$\begin{array}{l} \texttt{let} \; \{ \; \cdots ; \; \texttt{x} = \texttt{let} \; \{ \; \vec{b} \; \} \; \texttt{in} \; e_0 \; ; \cdots \; \} \; \texttt{in} \; e_1 \; \rightarrow \\ \qquad\qquad \texttt{let} \; \{ \; \cdots ; \; \vec{b} \; ; \; \texttt{x} = e_0 \; ; \cdots \; \} \; \texttt{in} \; e_1 \end{array}$$

These rules assume suitable variable renaming to ensure no variable shadowing. To illustrate, the definition of falseCase becomes

```
falseCase t n =
  let {x0 = tri x1 (+); x1 = 1 x2; x2 = n (-)} in n x0
```

It is easy to see the number and length of applications in an expression graph. For example, falseCase contains four applications and its longest application, tri x1 (+), has length three.

2.7 Template code

Each function definition is now of the form

$$f \; x_0 \cdots x_n \;\; = \texttt{let} \; \{ \; v_0 = e_0 \, ; \; \cdots \; ; \; v_m = e_m \; \} \; \texttt{in} \; e$$

where all expressions are flat (and the list of let-bindings may be empty). This is very close to the *template code* that can be actually executed natively by the

Reduceron. We shall define template code as a Haskell data type, paving the way for an executable semantics to be defined in Section 3. To highlight these semantic definitions, and to distinguish them from F-lite code, we prefix them using the '>' symbol.

In template code, a program is defined to be a list of templates.

```
> type Prog = [Template]
```

A template represents a function definition. It contains an *arity*, a *spine application* and a list of *nested applications*.

```
> type Template = (Arity, App, [App])
> type Arity = Int
```

The spine application holds the let-body of a definition's expression graph, and the nested applications hold the let-bindings. Applications are flat and are represented as a list of *atoms*.

```
> type App = [Atom]
```

An atom is a small, tagged piece of non-recursive data.

```
> data Atom =
>    FUN Arity Int  -- Function with arity and address
>  | ARG Int        -- Reference to a function argument
>  | PTR Int        -- Pointer to an application
>  | CON Arity Int  -- Constructor with arity and index
>  | INT Int        -- Integer literal
>  | PRI String     -- Primitive function name
>  | TAB Int        -- Case table
```

After applying transformations described in Sections 2.3 to 2.6, F-lite programs can be compiled to the Reduceron template code by the compilation scheme defined in Figure 2. The following paragraphs describe, less formally, how such programs are compiled to template code.

Functions. Given a list of function definitions

$$f_0\ \vec{x}_0\ =\ e_0,\ \cdots,\ f_n\ \vec{x}_n\ =\ e_n$$

each function identifier f_i occurring in $e_0 \cdots e_n$ is translated to an atom `FUN` $\#f\ i$, where $\#f$ is the arity of function f.

Arguments. In each definition $f\ x_0 \cdots x_n\ =\ e$, each variable x_i occurring in e is translated to an atom `ARG` i.

Let-bound variables. In each expression graph

$$\texttt{let \{ } x_0 = e_0\ ; \cdots ;\ x_n = e_n \texttt{ \} in } e$$

each x_i occurring in $e, e_0 \cdots e_n$ is translated to an atom `PTR` i.

Integers, primitives and constructors. An integer literal n, a primitive p and a constructor C_i are translated to atoms `INT` n, `PRI` p and `CON` $\#C_i\ i$ respectively.

$$
\begin{aligned}
D[\![\; f_i\, x_0 \cdots x_n \;= \texttt{let \{}\; v_0 = e_0 \,;\, \cdots \,;\; v_m = e_m \;\texttt{\} in}\; e \;]\!] \;&=\; (n+1, A_\sigma[\![e]\!], [A_\sigma[\![e_0]\!], \cdots, A_\sigma[\![e_m]\!]]) \\
\textbf{where} \quad & \\
\sigma \;&=\; [x_0 \mapsto \texttt{ARG}\; 0, \cdots, x_n \mapsto \texttt{ARG}\; n, \\
&\quad\;\; v_0 \mapsto \texttt{PTR}\; 0, \cdots, v_m \mapsto \texttt{PTR}\; m] \\
\\
A_\sigma[\![\; e_0 \cdots e_n \;]\!] \;&=\; [E_\sigma[\![e_0]\!], \cdots, E_\sigma[\![e_n]\!]] \\
\\
E_\sigma[\![\; n \;]\!] \;&=\; \texttt{INT}\; n \\
E_\sigma[\![\; p \;]\!] \;&=\; \texttt{PRI}\; p \\
E_\sigma[\![\; v \;]\!] \;&=\; \sigma\; v \\
E_\sigma[\![\; C_i \;]\!] \;&=\; \texttt{CON}\; \#C_i\; i \\
E_\sigma[\![\; f_i \;]\!] \;&=\; \texttt{FUN}\; \#f_i\; i \\
E_\sigma[\![\; \langle f_i, f_{i+1}, \cdots, f_j \rangle \;]\!] \;&=\; \texttt{TAB}\; i
\end{aligned}
$$

Fig. 2. Compilation scheme from transformed F-lite programs to the Reduceron template code. Scheme *D* applies to function definitions; scheme *A* applies to applications; scheme *E* applies to atomic expressions; and σ ranges over mappings from variables to atoms.

Case tables. Given a list of function definitions

$$f_0\ \vec{x}_0 \;=\; e_0,\ \cdots,\ f_n\ \vec{x}_n \;=\; e_n$$

each case table <f_i, $\cdots f_j$> occurring in $e_0 \cdots e_n$ is translated to an atom `TAB` *i*. We arrange that the functions in each case table are defined contiguously in template code.

Example. Consider the following program involving the `tri` function.

```
main          = let { } in tri 5
tri       n   = let { x = n (<=) } in 1 x <falseCase, trueCase> n
falseCase t n = let { x0 = tri x1 (+); x1 = 1 x2; x2 = n (-) } in n x0
trueCase  t n = let { } in 1
```

The template code for the above program is as follows.

```
> tri5 :: Prog
> tri5 = [ (0, [FUN 1 1, INT 5], [])
>        , (1, [INT 1, PTR 0, TAB 2, ARG 0],
>              [[ARG 0, PRI "(<=)"]])
>        , (2, [ARG 1, PTR 0],
>              [[FUN 1 1, PTR 1, PRI "(+)"],
>               [INT 1, PTR 2],
>               [ARG 1, PRI "(-)"]])
>        , (2, [INT 1], []) ]
```

Note that each function definition is mapped to a template represented as a triple. The first component of each template is the arity of the function. The second and third components represent the function's let-body and let-bindings respectively. Functions with no let-bindings are mapped to templates with an empty third component. Functions, applications and arguments are all referred to by position. For example, the atom `FUN 1 1` in the first template represents a call to a function of arity 1 found at index 1 in the list of templates (in this case, a call to the `tri` function). And the atom `PTR 2` in the third template is a pointer to the application at index 2 in the list of let-bound applications (in this case, the application named `x2`).

3 Operational semantics

This section defines a *small-step operational semantics* for the Reduceron. There are two main reasons for presenting a semantics: (1) To define precisely how the Reduceron works; and (2) to highlight the low-level parallelism present in graph reduction that is exploited by the Reduceron. We have found it very useful to encode the semantics directly in Haskell: before we commit to a low-level implementation, we can assess the complexity and performance of different design decisions and optimisations.

At the heart of the semantic definition is the small-step state transition function

```
> step :: State -> State
```

where the state is a 4-tuple comprising a program, a heap, a reduction stack and an update stack.

```
> type State = (Prog, Heap, Stack, UStack)
```

The heap is modelled as a list of applications, and can be indexed by a heap-address

```
> type Heap = [App]
> type HeapAddr = Int
```

An element on the heap can be modified using the update function

```
> update :: HeapAddr -> App -> Heap -> Heap
> update i a as = take i as ++ [a] ++ drop (i+1) as
```

The reduction stack is also modelled as a list of nodes, with the top stack element coming first and the bottom element coming last

```
> type Stack = [Atom]
> type StackAddr = Int
```

There is also an update stack

```
> type UStack = [Update]
> type Update = (StackAddr, HeapAddr)
```

The meaning of a program p is defined by `run` p, where

```
> run :: Prog -> Int
> run p = eval initialState
>   where initialState = (p, [], [FUN 0 0], [])

> eval (p, h, [INT i], u) = i
> eval s = eval (step s)
```

The initial state of the evaluator comprises a program, an empty heap, a singleton stack containing a call to `main` and an empty update stack. The `main` template has arity 0 and we arrange so that it is the template at address 0. To illustrate, `run tri5` yields `15`. In the following sections, the central `step` function is defined.

3.1 Primitive reduction

The `prim` function applies a primitive function to two arguments supplied as fully evaluated integers.

```
> prim :: String -> Atom -> Atom -> Atom
> prim "(+)" (INT n) (INT m) = INT (n+m)
> prim "(-)" (INT n) (INT m) = INT (n-m)
> prim "(<=)" (INT n) (INT m) = bool (n<=m)
```

The comparison primitive returns a boolean value. Both boolean constructors have arity 0; `False` has index 0 and `True` has index 1.

```
> bool :: Bool -> Atom
> bool False = CON 0 0
> bool True = CON 0 1
```

3.2 Normal forms

The number of arguments demanded by an atom on top of the reduction stack is defined by the `arity` function.

```
> arity :: Atom -> Arity
> arity (FUN n i) = n
> arity (INT i) = 1
> arity (CON n i) = n+1
> arity (PRI p) = 2
```

To reduce an integer, the evaluator demands one argument as shown in rewrite rule (2). And to reduce a constructor of arity n, the evaluator requires $n+1$ arguments (the constructor's arguments *and* the case table) as shown in rewrite rule (5).

The arity of an atom is only used to detect when a *normal form* is reached. A normal form is an application of length n whose first atom has arity $\geqslant n$.

Some functions, such as case-alternative functions, are statically known *never* to be partially applied, so they cannot occur as the first atom of a normal form. Such a function, say with address n, can be represented by the atom `FUN` 0 n.

3.3 Step-by-step reduction

There is one reduction rule for each possible type of atom that can appear on top of the reduction stack.

Unwinding. If the top of the reduction stack is a pointer x to an application on the heap, evaluation proceeds by *unwinding*: copying the application from the heap to the reduction stack where it can be reduced. We must also ensure that when evaluation of the application is complete, the location x on the heap can be updated with the result. So we push onto the update stack the heap address x and the current size of the reduction stack.

```
> step (p, h, PTR x:s, u) = (p, h, h!!x ++ s, upd:u)
>   where upd = (1+length s, x)
```

Updating. Evaluation of an application is known to be complete when an argument is demanded whose index is larger than n, the difference between the current size of the reduction stack and the stack address of the top update. If this condition is met, then a normal form of arity n is on top of the reduction stack and must be written to the heap.

```
> step (p, h, top:s, (sa,ha):u)
>   | arity top > n = (p, h', top:s, u)
>   where
>     n  = 1+length s - sa
>     h' = update ha (top:take n s) h
```

Integers and primitives. Integer literals and primitive functions can be reduced as described in Section 2.3, but see Sections 5.2 and 5.3 for refinements of this approach.

```
> step (p, h, INT n:x:s, u) = (p, h, x:INT n:s, u)
> step (p, h, PRI f:x:y:s, u) = (p, h, prim f x y:s, u)
```

Constructors. The Scott-encoded constructor applications are reduced by indexing a case table, as described in Section 2.4.

```
> step (p, h, CON n j:s, u) = (p, h, FUN 0 (i+j):s,u)
>   where TAB i = s!!n
```

There is insufficient information available to compute the arity of the case-alternative function at address `i+j`. However, an arity of zero can be used because a case-alternative function is statically known not to be partially applied (Section 3.2).

Function application. To apply a function f of arity n, $n+1$ elements are popped off the reduction stack, the spine application of the body of f is instantiated and pushed onto the reduction stack, and the remaining applications are instantiated and appended to the heap.

```
> step (p, h, FUN n f:s, u) = (p, h', s', u)
>   where
>     (arity, spine, apps) = p !! f
>     h' = h ++ map (instApp s h) apps
>     s' = instApp s h spine ++ drop arity s
```

Instantiating a function body involves replacing the formal parameters with arguments from the reduction stack and turning relative pointers into absolute ones.

```
> instApp :: Stack -> Heap -> App -> App
> instApp s h = map (inst s (length h))
> inst :: Stack -> HeapAddr -> Atom -> Atom
> inst s base (PTR p) = PTR (base + p)
> inst s base (ARG i) = s !! i
> inst s base a = a
```

4 Implementation

We now refine the semantic definition to an actual implementation that runs on an FPGA. Specifically, our target is a mid-range Xilinx Virtex-5 released in 2008. Our guiding design principle is to perform as much reduction as possible in each clock-cycle. Our implementation performs each semantic reduction rule in *a single clock-cycle*, and clocks around 100 MHz, a modest but respectable frequency for processor-like FPGA designs.

4.1 Low-level parallelism

Below we motivate three main opportunities for parallelism that we exploit in our implementation.

Parallel memories. The state of the reduction machine comprises four independent memory regions: the program, the heap, the reduction stack and the update stack. Most reduction rules refer to and modify more than one memory region. For example, the reduction rule for unwinding writes to both the reduction stack and the update stack. If the four memory regions are implemented as four separate memory units, then they can be accessed in *parallel*, avoiding contentions that would arise if they were all stored in a single memory unit.

Wide memories. Many of the reduction rules involve transferring applications to and from memory. If a memory only allows one atom to be accessed at a time, transferring a single application involves *several* memory accesses. If memories are wide enough to allow a whole application to be accessed at a time, transferring an application needs only a *single* memory access.

Parallel instantiation. The reduction rule for function application involves instantiating each application in a function body and appending it to the heap. Each atom in an application can be instantiated in parallel, as indicated by the use of `map` in the definition of `instApp`. The wide heap then allows the instantiated application to be written in one memory access. Further, each application in a function body can also be instantiated in parallel, as indicated by the use of `map` in the semantic rule for function application. If more than one application must be appended to the heap, these applications are instantiated in parallel. All this is made possible by imposing bounds on the structure of templates as discussed in Sections 4.2–4.4.

4.2 Bounded template instantiation

Maximum application length. Ideally, we would have a wide enough data bus to transfer *any* entire application in one go. However, this is an impossibility without some upper bound on the length of an application. Therefore, we introduce a bound, *MaxAppLen*, on the number of atoms that can occur in an application.

To deal with an application whose length is larger than *MaxAppLen*, we split it into two or more smaller ones. For example, if *MaxAppLen* is 3, the application $f\ a\ b\ c\ d\ e$ can be bracketed $((f\ a\ b)\ c\ d)\ e$ resulting in three applications rather than one.

An alternative way to bound application length is to split applications into chunks that are aligned contiguously in memory, with the final chunk especially tagged by an end-marker. This approach (Naylor & Runciman, 2008) is more efficient in some cases, but it cannot be expressed as a core-language transformation.

Maximum spine length. Spine applications are special because, during function application, they are written to the stack, not the heap. So it is fine for spine applications to have a different maximum length: *MaxSpineLen*.

Maximum applications per template. Ideally, all applications in a template would be instantiated in parallel. To allow for such an implementation, we introduce a bound, called *MaxAppsPerBody*, on the maximum number of applications that can occur in a template body. To deal with templates containing more applications than *MaxAppsPerBody*, we employ a technique called *template splitting*.

4.3 Template splitting

Example. We introduce template splitting by example. Consider the following template, representing the `falseCase` function occurring in the `tri5` program defined in Section 2.7.

```
(2, [ARG 1, PTR 0]                   -- Spine
  , [ [FUN 1 1, PTR 1, PRI "(+)"]    -- Application 1
    , [INT 1, PTR 2]                 -- Application 2
    , [ARG 1, PRI "(-)"] ] )         -- Application 3
```

It contains one spine application and three nested applications. If *MaxAppsPerBody* is two then this template is split into two sub-templates. The first sub-template

```
(0, [FUN 0 4]                    -- Intermediate spine
  , [ [FUN 1 1, PTR 1, PRI "(+)"]    -- Application 1
    , [INT 1, PTR 2] ] )             -- Application 2
```

replaces the original template in the `tri5` program. The second sub-template is appended to the program at the next free program address: address four in the case of the `tri5` program.

```
(2, [ARG 1, PTR (-2)]                -- Spine
  , [ [ARG 1, PRI "(-)"] ])          -- Application 3
```

```
split :: Int -> Template -> [Template]
split f t@(arity, spine, apps)
  | length apps < maxAppsPerBody = [t]
  | otherwise =
      (0, [FUN 0 f], take maxAppsPerBody apps)
    : split (f+1) (arity, spine, drop maxAppsPerBody apps')
  where
    apps' = map (map decPTR) apps

decPTR :: Atom -> Atom
decPTR (PTR n) = PTR (n - maxAppsPerBody)
decPTR a = a
```

Fig. 3. An application `split f t` splits a template `t` into a list of sub-templates, where `f` is the next unoccupied template address. The final sub-template returned replaces the original un-split template, and the other sub-templates are appended to the program beginning at address `f`.

The spine of the first sub-template is simply a call to the second sub-template. There are three important points to note:

- The first sub-template contains three applications, which is still larger than the bound *MaxAppsPerBody*. However, at the implementation level, we do not count a spine application of the form [FUN 0 *f*] as an application: It can be interpreted simply as 'jump to template *f*', and does not entail any heap or stack accesses.
- In the second sub-template, each atom of the form PTR *n* is replaced by PTR $(n-2)$ to account for the fact that instantiation of the first sub-template would have increased the size of the heap by two.
- The arity of the first sub-template is set to zero: No elements are popped from the stack since they may be required by the second sub-template.

Template splitting algorithm. In the general case, template splitting is performed by the `split` function defined in Figure 3. To illustrate, consider the following function body where `f`, `g` and `h` are function identifiers, and `x` and `y` are argument variables

```
f x (g (h y) x)
```

Supposing that *MaxAppsPerBody* is 2, then this body will be implemented as two templates: one containing the applications of `h` and `g` and another containing the application of `f`. Two templates mean that two reduction steps are required to instantiate the body.

This splitting method is rather simplistic, and there may be a room for improvement. One alternative would be to abstract out the second argument of `f` by introduction of

```
i x y = g (h y) x
```

allowing the following equivalent but smaller body

```
f x (i x y)
```

Table 1. *Effect of application-length, spine-length and applications-per-body on reduction count and heap usage*

Parameter	Bound	Reductions	Heap
MaxAppLen	2	1.00	1.00
	3	0.84	1.00
	4	0.83	1.30
	5	0.82	1.57
	6	0.82	1.89
MaxSpineLen	2	1.00	1.00
	3	0.82	0.76
	4	0.76	0.67
	5	0.71	0.60
	6	0.70	0.57
MaxAppsPerBody	1	1.00	1.00
	2	0.89	1.00
	3	0.85	1.00
	4	0.85	1.00

This smaller body can be implemented as a single template, and instantiated in a single reduction step. In the case where the second argument of `f` is never demanded, one reduction step is saved. In the case where the second argument of `f` is demanded, a reduction of `i` is required, and overall no reduction step is saved and neither is any incurred.

So splitting up function bodies at the template level is not the only option. We could split function bodies at the F-lite combinator level; as illustrated above, this could lead to improved performance.

4.4 Choosing the bounds

We must choose the values of the bounds *MaxAppLen*, *MaxSpineLen* and *MaxAppsPerBody* carefully: making them too low prevents useful parallelism; making them too high wastes resources. Our choices are informed by experiment. Table 1 shows the performance effect of varying each parameter in turn – non-varying parameters are effectively defined as infinity. The reduction count and heap usage figures are normalised across the varying parameter and averaged across a range of benchmark programs (see Section 6.1). The measurements are obtained by using a PC implementation of operational semantics. The reduction count represents the number of times that the `step` function is applied in the definition of `eval`.[1]

The chosen bounds are: *MaxAppLen* $= 4$, *MaxSpineLen* $= 6$ and *MaxAppsPer Body* $= 2$. The measurements suggest that a *MaxAppLen* of three is preferable to four due to better heap usage; the choice of four is motivated by another implementation

[1] Constructor reductions are not counted, anticipating the optimisation presented in Section 4.7.

Table 2. *Size and type of each parallel memory unit*

Memory unit	Element	Bits per element	Elements
Program	`Template`	234	1 k
Heap	`App`	77	32 k
Reduction stack	`Atom`	18	8 k
Update stack	`Update`	28	4 k
Case-table stack	`Atom`	18	4 k
Copy space	`App`	77	16 k

parameter – the arity limit – introduced in Section 4.5. A *MaxSpineLen* of five would not be much worse than six, but the choice of six does not cost much extra at the implementation level. A *MaxAppsPerBody* of two is motivated by the fact that three would not be much better and that two fits nicely with the dual-port memories available on the FPGA.

4.5 Memory layout

Our Xilinx Virtex-5 FPGA contains 296 dual-port block RAMs, each with a capacity of 18 kilobits giving a total on-chip RAM capacity of 5,328 kilobits. Each block RAM is dual-port allowing two independent accesses per clock-cycle. The data-bus and address-bus widths of each block-RAM are configurable. Possible configurations include 1 k-by-18 bit and 16 k-by-1 bit, and a range of possibilities in-between. Two 18 kilobits block RAMs can be merged to give further possible configurations ranging from 1 k-by-36 bit to 32 k-by-1 bit.

For simplicity, our implementation uses FPGA block RAMs only; no off-chip RAMs are used. This represents a tight constraint on the amount of memory available to the implementation. (The possibility of introducing off-chip memories is discussed in Section 7.)

Memory structure. The parallel memory units, each built out of block RAMs, are listed in Table 2 along with their capacities and the type of element stored at every addressable location. Note that there are uniform sizes for every program template and for every heap application. The two memory units at the bottom of the table are introduced in Sections 4.7 and 4.8 respectively.

Wide memories. The wide heap memory is implemented by concatenating the data-busses of 77 32 k-by-1 bit block RAMs and merging their address-busses. This is done on both ports of each block RAM, making a *dual-port heap*. Similarly, the wide program memory is implemented using 13 1 k-by-18 bit block RAMs, but this time the dual-port capability is not needed.

Stack memories. We store the top N stack elements in special-purpose stack registers. In any given clock-cycle, the stack implementation allows: the top N elements to be observed; *and* up to N elements to be popped off; *and* up to N elements to be pushed on. If pushing and popping occur in the same clock-cycle, the pop is performed before the push. Simultaneous access to the top N elements of the stack is achieved by a crossbar switch: it requires over 2,000 logic gates, but this is less than 1% of our FPGA's logic-gate capacity. There is a lot of parallelism in a crossbar, so the investment is worth it. Further hardware-level implementation details of the stack implementation are available in Naylor (2009b).

Arity limit. The stack implementation is parameterised by N, but requires N to be a power of two. For the update stack, N is defined to be 1 since reading and writing multiple values is of no benefit. For the reduction stack, there are three considerations to take into account, bearing in mind the aim of single-cycle reduction: (1) only the top N stack elements are available in any clock-cycle, hence the maximum number of arguments that be taken by a function is $N - 1$; (2) the maximum length of a partially applied function application, or normal-form, is therefore $N - 1$; and (3) the choice of N should allow a normal form of length $N - 1$ to be written onto the heap in a single clock-cycle. As two applications of length *MaxAppLen* can be written to the dual-port heap per clock-cycle, and *MaxAppLen* is four, a sensible choice for N is eight, since a normal form of length seven can be bracketed perfectly into two applications of length four.

To deal with functions taking more than $N-1$ arguments, an abstraction algorithm can be used (Turner, 1979). We have developed a minor variant (Naylor, 2009a) of an abstraction algorithm based on director strings (Dijkstra, 1980; Kennaway & Sleep, 1988), which uses a more coarse-grained combinator set than Turner's algorithm. However, we note that none of the benchmark programs used in this paper (Section 6) exceeds the arity limit of 8.

4.6 One reduction per clock-cycle

Heap and program memory units are designed to have the following two properties:

- **Property 1**. If a memory location x is read in clock-cycle n, the value at address x becomes available on the memory's data bus on clock-cycle $n + 1$.
- **Property 2**. If a value is written to memory location x in clock-cycle n, the new value at address x is not apparent until clock-cycle $n + 1$.

The top stack elements are always observable without any clock-cycle delay. Now we show how each reduction rule in the semantics can be performed in a single clock-cycle, with reference to the following two invariants.

- **Invariant 1**. If the top of the reduction stack is of the form `PTR` x then the application at heap address x is currently available on the heap memory's data bus.
- **Invariant 2**. If the top of the reduction stack is of the form `FUN` n f then the template at program address f is currently available on the program memory's data bus.

Unwinding. The top of the reduction stack has the form `PTR` x. So the application currently on the heap's data bus, say *app*, is the application at heap address x (Invariant 1). The following memory transactions are performed in parallel in a single clock-cycle:

- The application *app* is pushed onto the reduction stack.
- An update (n, x) is pushed onto the update stack, where n is the size of the reduction stack before modification.
- The first atom of *app* is the new top of the reduction stack and is used to look up heap and program memory in order to maintain Invariants 1 and 2.

Updating. The update stack's data bus is used to determine if an update is required, and if so, at what heap address x. If an update is required, then a normal form is available on the reduction stack's data bus. The following memory transactions are performed in parallel in a single clock-cycle:

- If the normal form has length less than or equal to four, it is written to the heap at address x.
- If the normal form has length larger than four, it is bracketed into two applications of maximum length four, one of which is written to the heap at address x, and the other is appended to the heap.
- The top element of the update stack is popped.
- A program lookup is performed to preserve Invariant 2.

A heap lookup to preserve Invariant 1 is not necessary since the top of the reduction stack cannot possibly be of the form `PTR` x if an update is being performed. So updating requires at most two heap accesses, which can be done parallel, thanks to dual-port memory.

Integers, primitives and constructors. Each of these reduction rules involves a pure stack manipulation, and each straightforwardly consumes a single clock-cycle.

Function application. The top of the reduction stack has the form `FUN` n f. So the template of f, say t, is available on the data bus (Invariant 2). There are two cases to consider.

Case 1. If t contains a spine application of the form [`FUN` 0 f], then

- *up to two* nested applications in t are instantiated and appended to the heap;
- the atom `FUN` 0 f is written to the top of the reduction stack; and
- function f is looked up in program memory in order to preserve Invariant 2.

Case 2. If t is of some other form, then

- *zero or one* nested applications in t are instantiated and appended to the heap;
- the spine application in t is instantiated and written to the reduction stack; and
- the first element of the instantiated spine is used to look up heap and program memory to preserve Invariants 1 and 2.

Table 3. *Impact of optimisations on clock-cycle count across a range of programs*

Program	Baseline	+In-lining	+Case Stack	+Update Avoidance	+Infix Primitives	+PRS
Adjoxo	1.00	0.85	0.71	0.54	0.43	0.36
Braun	1.00	0.84	0.63	0.46	0.43	0.42
Cichelli	1.00	0.93	0.77	0.56	0.42	0.41
Clausify	1.00	0.79	0.59	0.48	0.41	0.41
CountDown	1.00	0.95	0.86	0.70	0.49	0.31
Fib	1.00	1.28	1.21	0.96	0.75	0.35
KnuthBendix	1.00	0.81	0.63	0.48	0.43	0.40
Mate	1.00	0.83	0.67	0.50	0.43	0.40
MSS	1.00	0.92	0.84	0.61	0.38	0.24
OrdList	1.00	0.73	0.55	0.42	0.42	0.42
PermSort	1.00	0.77	0.62	0.48	0.42	0.42
Queens	1.00	0.75	0.68	0.51	0.40	0.21
Queens$_2$	1.00	0.82	0.67	0.55	0.50	0.50
SumPuz	1.00	0.95	0.80	0.60	0.50	0.48
Taut	1.00	0.90	0.70	0.56	0.51	0.50
While	1.00	0.93	0.77	0.58	0.50	0.49
Average	1.00	0.88	0.74	0.57	0.47	0.40

In Case 1, a heap lookup to preserve Invariant 1 is not required: the top of the stack is known to be a `FUN`, not a `PTR`. Thus, in each case at most two heap access are required.

4.7 The case-table stack

Constructor reduction modifies only the top element of the reduction stack by adding the index of the constructor to the address of a case table. This addition is *almost* cheap enough to be implemented in combinatorial logic (i.e. in zero clock-cycles) without lengthening the critical path of the circuit. The problem is that the case table must be fetched from *a variable position* on the stack. This requires a multiplexer, making the combinatorial logic more expensive.

To solve this problem, we introduce a new stack memory to store case tables. When unwinding an application containing a case table, the case table is pushed onto the *case-table stack*. When performing constructor reduction, the case table of interest is always in the same position: the *top* of the case-table stack.

Tables 3 and 4 show the impact of various optimisations on clock-cycle count and heap usage across a range of benchmark programs. Together with the in-lining strategy defined in Section 2.5, the case-table optimisation results in significant performance gains on average. The other optimisations in Tables 3 and 4 are introduced in Section 5. Note that in the Fib program, inlining has a slight negative effect: the body of the recursive-case of `fib` grows to contain two applications both of which need to be split, increasing instantiation time and unwinding time. Without

Table 4. *Impact of optimisations on heap usage across a range of programs*

Program	Baseline	+In-lining	+Case Stack	+Update Avoidance	+Infix Primitives	+PRS
Adjoxo	1.00	0.80	0.80	0.80	0.49	0.41
Braun	1.00	0.93	0.93	0.93	0.88	0.88
Cichelli	1.00	0.97	0.97	0.97	0.36	0.33
Clausify	1.00	0.59	0.59	0.59	0.42	0.42
CountDown	1.00	0.97	0.97	0.97	0.53	0.33
Fib	1.00	2.33	2.33	2.33	2.00	0.33
KnuthBendix	1.00	0.66	0.66	0.66	0.58	0.49
Mate	1.00	0.45	0.45	0.45	0.29	0.25
MSS	1.00	1.00	1.00	1.00	0.51	0.03
OrdList	1.00	0.67	0.67	0.67	0.67	0.67
PermSort	1.00	0.77	0.77	0.77	0.69	0.69
Queens	1.00	0.54	0.54	0.54	0.39	0.11
Queens$_2$	1.00	0.92	0.92	0.92	0.77	0.73
SumPuz	1.00	1.06	1.06	1.05	0.74	0.63
Taut	1.00	0.99	0.99	0.99	0.90	0.87
While	1.00	0.95	0.95	0.95	0.81	0.80
Average	1.00	0.92	0.92	0.92	0.69	0.50

inlining, no applications are split because the long applications remain in the spine position. There is scope for cleverer inlining.

4.8 Garbage collection

Our implementation employs a simple two-space stop-and-copy garbage collector (Jones & Lins, 1996). Although a two-space collector may not make the best use of limited memory resources, it does have the attraction of being easy to implement. In particular, the algorithm is easily defined iteratively so that no recursive call stack is needed. See Section 6.3 for some performance figures for garbage collection.

4.9 Hardware description

The Reduceron is described entirely in around 2,000 lines of York Lava (Naylor *et al.*, 2009), a hardware description language embedded in Haskell. A large proportion of the description deals with garbage collection and the bit-level encoding of template code; the actual reduction rules account for less than 400 lines.

The Reduceron description is quite different from other reported Lava applications. It combines *structural and behavioural* description styles. Behavioural description brings improved *modularity*: we associate each reduction rule with the memory transactions it performs, rather than associating each memory unit with all the memory transactions performed on it. So each reduction rule can be expressed in isolation.

The behavioural description language, called Recipe and included with York Lava, takes the form of a 300-line Lava library. It provides mutable variables, assignment statements, sequential and parallel composition, conditional and looping constructs and shared procedure calls. In addition, it uses the results of a simple timing analysis, implemented by abstract interpretation, to enable optimisations.

4.10 Synthesis results

Synthesising our implementation on a Xilinx Virtex-5 LX110T (speed-grade 1) yields an FPGA design using 14% of available logic slices and 90% of available block RAMs. The maximum clock frequency after place-and-route is 96 MHz. By comparison, Xilinx distributes a hand-optimised RISC soft-processor called the MicroBlaze (Xilinx, 2009) that clocks at 210 MHz on the same FPGA. As the Reduceron performs a lot of computation per clock-cycle and is *not pipelined* (the Microblaze has five pipeline stages), 96 MHz seems respectable. However, whereas the Microblaze is a 32-bit processor, the Reduceron only processes 18-bit atoms (18 bits are sufficient to address the entire heap stored in block RAM); increasing the atom size to 32 bits may affect the Reduceron's maximum clocking frequency. See Section 6.3 for a performance comparison of the Reduceron executing template code and the Microblaze executing conventionally compiled code.

5 Optimisations

This section presents several optimisations, defined by a series of progressive modifications to the semantics defined in Section 3. A theme of this section is the use of *cheap dynamic analyses* to improve performance.

5.1 Update avoidance

Recall that when evaluation of an application on the heap is complete, the heap is updated with the result to prevent repeated evaluation. There are two cases in which such an update is unnecessary: (1) The application is *already evaluated*, and (2) the application is *not shared*, so its result will never be needed again. On the Reduceron, each update consumes a clock-cycle, so avoiding some unnecessary updates will improve performance.

We identify non-shared applications at *run-time*, by *dynamic* analysis. Argument and pointer atoms are extended to contain an extra boolean field.

```
> data Atom = ··· | ARG Bool Int | PTR Bool Int | ···
```

An argument is tagged with `True` exactly if it is referenced more than once in the body of a function. A pointer is tagged with `False` exactly if it is a *unique pointer*; that is, it points to an application that is not pointed to directly by any other atom on the heap or reduction stack. There may be multiple pointers to an application containing a unique pointer, so the fact that a pointer is unique is, on its own, not enough to infer that it points to a non-shared application. To identify non-shared applications, we maintain the following invariant.

- **Invariant 3**. A unique pointer occurring on the reduction stack points to a non-shared application.

A pointer that is not unique is referred to as *possibly-shared.*

Unwinding. The reduction rule for unwinding becomes

```
> step (p, h, PTR sh x:s, u) = (p, h, app++s, upd++u)
>   where
>     app = map (dashIf sh) (h!!x)
>     upd = [(1+length s, x) | sh && red (h!!x)]
```

If the pointer on top of the stack is possibly-shared, then the application is *dashed* before being copied onto the stack by marking each atom it contains as possibly-shared. This has the effect of propagating sharing information through an application

```
> dashIf sh a = if sh then dash a else a

> dash (PTR sh s) = PTR True s
> dash a = a
```

If the pointer on top of the stack is unique, the application it points to must be non-shared according to Invariant 3. An update is only pushed onto the update stack if the pointer is possibly-shared and the application is *reducible.* An application is reducible if it is saturated or its first atom is a pointer.

```
> red :: App -> Bool
> red (PTR sh i:xs) = True
> red (x:xs) = arity x <= length xs
```

Updating. When an update occurs, the normal-form on the stack is written to the heap. The normal-form may contain a unique pointer, but the process of writing it to the heap will duplicate it. Hence, the normal-form on the stack is dashed

```
> step (p, h, top:s, (sa,ha):u)
>   | arity top > n = (p, h', top:dashN n s, u)
>   where
>     n  = 1+length s - sa
>     h' = update ha (top:take n s) h

> dashN n s = map dash (take n s) ++ drop n s
```

It is unnecessary to dash the normal-form that is written to the heap, but there is no harm in doing so: The application being updated is possibly-shared, and a possibly-shared application will anyway be dashed when it is unwound onto the stack.

Function application. When instantiating a function body, shared arguments must be dashed as they are fetched from the stack

```
> inst s base (PTR sh p) = PTR sh (base + p)
> inst s base (ARG sh i) = dashIf sh (s!!i)
> inst s base a = a
```

Performance. Tables 3 and 4 show that update avoidance offers a significant run-time improvement. On average, 88% of all updates are avoided across the 16 benchmark programs. Just over half of these are avoided due to non-reducible applications, and just under half of them are avoided due to non-shared reducible applications. The average maximum update-stack usage drops from 406 to 11.

The dynamic sharing analysis is not perfect. Some opportunities for update avoidance are missed when a pointer is copied at run-time but the copy is thrown away before the application it points to is actually evaluated. A full reference-counting garbage collector would allow even more accurate sharing information to be maintained.

5.2 Infix primitive applications

Consider the steps needed to evaluate an application `(+)` e_0 e_1. Using the approach to primitive reduction of Section 3.1, the application is translated to e_1 (e_0 `(+)`) at compile-time. At run-time, we need at least the following steps:

1. An integer reduction is required after e_1 is evaluated (to perform rewrite rule (2) defined in Section 3.1).
2. An unwind reduction is required to fetch the application e_0 `(+)` from the heap.
3. A second integer reduction is required after e_0 is evaluated.
4. A primitive reduction is required to apply `(+)` to the two evaluated arguments.

Requiring these four steps to reduce any primitive application is quite expensive. In this section, we introduce a new method for handling primitive applications that does not lead to so many reduction steps.

New method. For every binary primitive function p, we introduce a new primitive $*p$, a version of p that expects its arguments flipped

```
> prim ('*':p) n m = prim p m n
```

Any primitive function p can be flipped

```
> flip ('*':p) = p
> flip p = '*':p
```

Now we translate binary primitive applications by the rule

$$p\ m\ n \quad \rightarrow \quad m\ p\ n \tag{6}$$

In place of the existing reduction rules for primitives and integers, we define

```
> step (p, h, INT m:PRI f:INT n:s, u) =
>   (p, h, prim f m n:s, u)
> step (p, h, INT m:PRI f:x:s, u) =
>   (p, h, x:PRI (flip f):INT m:s, u)
```

If both arguments are already evaluated, the primitive is applied. If only the first argument is evaluated, then the arguments are swapped and the primitive is flipped.

Note that compilation rule (6) could just as sensibly be

$$p\ m\ n \quad \rightarrow \quad n\ {*p}\ m \tag{7}$$

In the interest of efficiency, the choice between (6) and (7) is informed for each primitive application by compile-time knowledge of whether m or n is expected to be already-evaluated.

Example. A primitive application of the form `(+)` e_0 e_1 is now translated to e_0 `(+)` e_1 at compile-time. And at run-time, we now only need the following steps:

1. An integer reduction is required after e_0 is evaluated.
2. A primitive reduction is required after e_1 is evaluated.

We have saved two reduction steps for every primitive application. Also note that e_0 `(+)` e_1 comprises one application whereas e_1 (e_0 `(+)`) comprises two, so the former is cheaper to instantiate.

Results. Tables 3 and 4 show run-time and heap-usage improvements brought by the new approach.

5.3 Speculative evaluation of primitive redexes

Consider evaluation of the expression `tri 5`. Application of `tri` yields the expression

```
case (<=) 5 1 of
  { False -> (+) (tri ((-) 5 1)) 5 ; True -> 1 }
```

which contains two *primitive redexes*: `(<=) 5 1` and `(-) 5 1`. This section introduces a technique called primitive-redex speculation (PRS), in which such redexes are evaluated during function body instantiation. For example, application of `tri` instead yields

```
case False of { False -> (+) (tri 4) 5 ; True -> 1 }
```

The benefit is that primitive redexes need not be constructed in memory, nor fetched again when needed. Even if the result of a primitive redex is *not* needed, reducing it is no more costly than constructing it. We identify primitive redexes at *run-time*, by *dynamic* analysis.

Register file. To support PRS, we introduce a *register file* to the reduction machine for storing the results of speculative reductions

```
> type RegFile = [Atom]
```

The body of a function may refer to these results as required

```
> data Atom = ··· | REG Bool Int
```

An atom of the form `REG` b i contains a reference i to a register, and a boolean field b that is true exactly if there is more than one reference to the register in the body of the function.

The instantiation functions `inst` and `instApp` are modified to take the register file r as an argument, and the following equation is added to the definition of `inst`

```
> inst s r base (REG sh i) = dashIf sh (r !! i)
```

Waves. The primitive redexes in a function body are evaluated in a series of *waves*. To illustrate, consider `(+) 1 ((+) 2 3)`. In the first wave of speculative evaluation, `(+) 2 3` would be reduced to `5`; in the second wave, `(+) 1 5` would be reduced to `6`.

More specifically, a wave is a list of *independent* primitive redex *candidates*. A primitive redex candidate is an application that may turn out at run-time to be a primitive redex. Specifically, it is an application of the form `[`a_0`, PRI` p`,` a_1`]`, where a_0 and a_1 are `INT`, `ARG` or `REG` atoms.

```
> type Wave = [App]
```

Templates are extended to contain a list of waves in which no application in a wave depends on the result of an application in the same or a later wave.

```
> type Template = (Arity, App, [App], [Wave])
```

Given the reduction stack, the heap, and a wave list, `prs` produces a possibly modified heap, and one result for each application in each wave

```
> prs :: Stack -> Heap -> [Wave] -> (Heap, RegFile)
> prs s h = foldl (wave s) (h, [])

> wave s (h,r) = foldl spec (h,r) . map (instApp s r h)
```

If a primitive redex candidate turns out to be a primitive redex at run-time, it is reduced and its result is appended to the register file. Otherwise, the candidate application is constructed on the heap, and a pointer to this application is appended to the register file

```
> spec (h,r) [INT m,PRI p,INT n] = (h, r++[prim p m n])
> spec (h,r) app = (h++[app], r++[PTR False (length h)])
```

Function application. Since applications in a function body may refer to the results in the PRS register file, PRS is performed before instantiation of the body. The new rule is

```
> step (p, h, FUN n f:s, u) = (p, h'', s', u)
>   where
>     (pop, spine, apps, waves) = p !! f
>     (h', r) = prs s h waves
>     s' = instApp s r h' spine ++ drop pop s
>     h'' = h' ++ map (instApp s r h') apps
```

The template-splitting technique outlined in Section 4.3 is modified to deal with waves of primitive redex candidates. Each wave is split into a separate template. If a wave contains more than *MaxAppsPerBody* applications, it is further split in order to satisfy the constraint.

Strictness analysis. PRS works well when recursive call sites sustain *unboxed* arguments.[2] For example, if a call to `tri` is passed an unboxed integer then, thanks to PRS, so too is the recursive call. However, if the initial call is passed a boxed expression, primitive redexes never arise, e.g. the outer call in `tri (tri 5)` is passed a pointer to an application, inhibiting PRS.

A basic strictness analyser in combination with the worker–wrapper transformation (Gill & Hutton, 2009) alleviates this problem. Each initial call to a recursive function is replaced with a call to a wrapper function. The wrapper applies a special primitive to force evaluation of any strict integer arguments before passing them on to the recursive worker.

Performance. Tables 3 and 4 show how PRS cuts run-time and heap-usage over the range of benchmark programs. On average, the maximum stack usage drops from 811 to 104, and 85% of primitive redex candidates turn out to be primitive redexes.

6 Comparative evaluation

This section evaluates the Reduceron in the context of previous and current work on functional language implementation.

6.1 Benchmark programs

The performance of the Reduceron is measured using a set of 16 benchmark programs. The programs, though small (the largest is of 551 lines), are diverse and fairly representative of functional programs in general. The programs are available online (Naylor *et al.*, 2009). Below are brief descriptions of each program.

Adjoxo (108 lines). An adjudicator for the game *noughts and crosses*. The input is a game position, and the output is one of the three values *Win*, *Draw* or *Loss* indicating the outcome with the best play for each of the players whose turn it might be. The method is the usual minimax recursive evaluation of completed game trees.

Braun (51 lines). A Braun tree is a balanced binary tree offering an efficient yet simple implementation of flexible arrays. The program tests the property that converting a list to a Braun tree and back again is equivalent to the identity function.

Cichelli (200 lines). Finds a perfect hash function for Haskell keywords. It uses a backtracking search to find an assignment of natural-number values to each letter that starts or ends a keyword such that hash values for keywords, computed as *start-value* + *end-value* + *length*, are unique and occupy a small integer range without gaps.

[2] An unboxed integer is an integer literal `INT` *n* as opposed to a pointer `PTR` *x* to an expression of type integer.

Clausify (132 lines). Puts propositional formulae in clausal form using a multi-stage transformation of formula-trees. It is almost a purely symbolic application, with hardly any arithmetic.

CountDown (120 lines). Hutton's solution to the countdown problem, a numbers game in which the aim is to construct arithmetic expressions satisfying certain constraints (Hutton, 2002).

Fib (10 lines). Computes the Nth number in the Fibonacci sequence using a simple but naive doubly recursive function definition. It is a purely arithmetic program involving no data structures at all.

KnuthBendix (550 lines). The Knuth–Bendix completion method tries to derive a convergent term-rewriting system for a given equational theory and symbol-weighting scheme. It is a typical symbolic computing application from computer algebra. The example input used in the program gives group-theoretic axioms from which 10 rewriting rules are derived.

Mate (393 lines). Solves chess end-game problems of the form '*P* to move and mate in *N*'. The method is a brute-force search in an explicit AND-OR game tree developing the given position to depth of 2N-1. Boards are represented by a square-piece association list for each player, where squares are coded as rank-file numeric pairs, so there is a fair amount of primitive arithmetic and comparison.

MSS (47 lines). Computes the maximum segment sum of a list of integers. It works by dividing the input list into all sub-lists, computing the sum of each and returning the maximum.

OrdList (46 lines). Checks the property that insertion of a number into an ordered list of numbers results in a list that is still ordered. Numbers are represented as peano numerals, so this is a purely symbolic program.

PermSort (39 lines). Enumerates the permutations of a list of numbers, and returns the first ordered permutation.

Queens (49 lines). Solves a programming problem made famous by Wirth: Place N queens on an $N \times N$ chess board so that no two queens occupy a common rank, file or diagonal. The solution involves backtracking, list processing and an inner recursive loop testing the safety of each candidate position for a new queen by primitive arithmetic comparisons with the coded positions of queens already in place.

Queens2 (62 lines). A purely symbolic solution to the N-queens problem which represents the board as a list of lists. It places a queen on one row at a time, maintaining a grid of threatened squares, and backtracks if a queen cannot be placed.

SumPuz (158 lines). A cryptarithmetic solver applied to several problem instances.

Taut (97 lines). A tautology checking program. The method is a brute-force evaluation for all possible boolean assignments to variables.

While (96 lines). A structural operational semantics for the While language (Neilson & Neilson, 2007) applied to an imperative program that naively computes the number of divisors of a given integer.

6.2 Comparison with the 2007 Reduceron

Compared to the Reduceron presented in Naylor & Runciman (2008), the implementation described in this paper reduces the number of clock-cycles required to run the benchmark programs by an average factor of 6.4. As the previous implementation clocks at 111 MHz, and the new one at 96 MHz on the same FPGA, the raw speed-up factor is 5.5. The gains are mainly due to the combined impact of improved case-expression compilation, single-cycle reduction and the optimisations listed in Tables 3 and 4. But another factor is that the new implementation performs *spineless* evaluation (Burn *et al.*, 1988). During function application, the spine of a function body is only written onto the stack, reducing heap contention and heap usage. The spine is only ever written to the heap during updating, and even then, only if it is a possibly shared normal-form. Spineless evaluation is also more modular in the sense that it allows function application to be conceptually separated from updating.

6.3 Comparison with conventionally compiled graph reduction

In this section we compare the run-time performance of the Reduceron against that of two conventional lazy functional language implementations: (1) The state-of-the-art Glasgow Haskell Compiler (GHC, version 7.0.3); and (2) our own F-lite-to-C compiler (FtoC).

The first aim of our comparison is to see how FPGA technology can fare against conventional PCs for the task of sequential graph reduction. We simply compare wall-clock times of programs running on the Reduceron against GHC-compiled programs running on a desktop PC. However, we must be careful not to infer too much from this comparison: The Reduceron is running on *reconfigurable* hardware but could in-principle be realised as a *specially fabricated* chip with a higher clock frequency.

The second aim of our comparison is to determine the performance benefit (if any) offered by the special architectural features of the Reduceron. One way to determine this is to configure the FPGA as a conventional processor and run the GHC-compiled programs on it; this way, both implementations are realised using the same circuit technology. Unfortunately, GHC does not yet target any

Table 5. *Performance of programs running on an Intel Core 2 Duo processor clocking at 3 GHz using (1) the GHC compiler and (2) the FtoC compiler. The acronym '%GC' denotes 'percentage of time spent garbage collecting'. FtoC-compiled programs have a 480 kilobyte heap*

	GHC -O2		FtoC		
Program	Time (s)	%GC	Time (s)	%GC	Slow-down Factor
Adjoxo	0.07	1.6	0.15	2.0	2.14
Braun	0.27	9.8	0.47	8.4	1.74
Cichelli	0.06	3.2	0.15	6.5	2.50
Clausify	0.20	20.0	0.39	12.4	1.95
CountDown	0.05	2.6	0.14	2.8	2.80
Fib	0.28	0.0	0.61	0.4	2.17
KnuthBendix	0.05	6.6	0.05	18.2	1.00
Mate	0.66	2.7	2.90	10.0	4.40
MSS	0.13	3.2	0.19	4.6	1.46
OrdList	0.61	45.3	0.51	1.4	0.83
PermSort	0.51	2.5	0.72	1.4	1.41
Queens	0.11	0.9	0.21	1.0	1.90
Queens2	0.47	15.7	0.57	5.8	1.21
SumPuz	0.80	4.4	1.64	3.9	2.05
Taut	0.16	34.8	0.20	2.7	1.25
While	0.08	2.7	0.18	1.0	2.25

FPGA environments. Therefore, we have developed our own compiler (FtoC) which generates standalone C code that can run on the MicroBlaze, a 32-bit RISC processor for FPGAs developed by Xilinx.

FtoC is an experimental compiler developed within a short period just for this paper. It only supports a first-order subset of F-lite; we have specialised higher order programs in a straightforward mechanical fashion. To deal efficiently with primitive arithmetic operations, FtoC supports a variant of primitive redex speculation based on *static analysis* (static PRS), discussed further in Section 6.9. Table 5 shows that FtoC-compiled programs typically run around a factor of two slower than GHC-compiled programs on a conventional PC. One exception is the Mate program, on which GHC's optimiser has a particularly significant effect.

Table 6 shows that the Reduceron is typically between 10 and 20 times faster than FtoC-compiled programs running on the MicroBlaze; we therefore predict that it would typically be between 5 and 10 times faster than GHC-compiled code running on the Microblaze. The three exceptions are Fib, Queens and KnuthBendix. In Fib and Queens, static PRS makes FtoC very effective; however, as shown in Table 8, the Reduceron can also benefit significantly from static PRS on these two programs. In KnuthBendix, FtoC benefits a lot from the specialisation of higher order functions, but so too would the Reduceron.

Note that FtoC-compiled programs running on the MicroBlaze have been limited to a 176 kilobyte heap: this is the maximum heap size we were able to obtain in a MicroBlaze setup where all memory is realised in block RAMs. The effect can be

Table 6. *Performance of programs running on a Virtex-5 FPGA using (1) the FtoC compiler targeting the MicroBlaze processor and (2) the Reduceron. FtoC-compiled programs have a 176 kilobyte heap and the Reduceron a 462 kilobyte heap*

	FtoC (210 MHz)		Reduceron (96 Mhz)		
Program	Time (s)	%GC	Time (s)	%GC	Speed-up factor
Adjoxo	3.98	5.2	0.39	0.5	10.21
Braun	14.85	34.1	0.74	6.7	20.06
Cichelli	5.03	15.6	0.41	1.3	12.27
Clausify	14.03	36.7	0.73	4.4	19.48
CountDown	3.49	7.6	0.19	1.4	18.36
Fib	21.77	0.8	2.52	0.1	8.64
KnuthBendix	1.60	52.1	0.19	14.7	8.42
Mate	118.18	40.8	6.74	3.0	17.53
MSS	8.22	12.8	0.69	0.1	11.91
OrdList	12.69	4.3	0.98	0.9	12.94
PermSort	18.83	4.9	1.62	1.3	11.62
Queens	6.84	2.2	1.00	0.2	6.84
Queens2	15.32	16.7	1.29	3.2	11.87
SumPuz	45.14	9.6	3.82	1.4	11.82
Taut	5.75	7.8	0.57	1.0	10.08
While	5.00	2.9	0.36	0.4	13.88

seen in the increased proportion of time spent on garbage collection, especially in KnuthBendix and Mate.

Analysing Tables 5 and 6 together, we deduce that on average GHC-compiled programs running on a 3 GHz PC are around 4–5 times faster than programs running on the 0.1 GHz Reduceron. Given the speed-up over our previous implementation of the Reduceron, we had hoped that the performance of our new implementation would approach that of the PC implementations. However, new GHC optimisations and the use of a 3 GHz Core 2 Duo instead of a 2.8 GHz Pentium-4 have significantly boosted the PC results. (The Dhrystone (Weicker, 1984) MIPS per MHz of the Core 2 Duo is almost twice that of the Pentium-4 (Longbottom, 2009).)

In summary, architectural features of the Reduceron do offer a significant performance boost, but not enough to fully absorb the difference in clock frequencies between PC and FPGA technologies.

6.4 Hand-reductions per clock-cycle

Table 7 shows *hand-reductions per clock-cycle*. A hand-reduction is the application of a function or a primitive function; it includes applications of functions introduced by case compilation, but does *not* include updating, unwinding, integer reduction, constructor reduction or applications of functions introduced by template splitting.

Table 7. *Hand-reductions per clock-cycle performed by the Reduceron*

Program	Hand-reductions per clock-cycle
Adjoxo	0.65
Braun	0.46
Cichelli	0.52
Clausify	0.54
CountDown	0.62
Fib	0.90
KnuthBendix	0.47
Mate	0.52
MSS	0.65
OrdList	0.43
PermSort	0.43
Queens	0.72
Queens2	0.40
SumPuz	0.46
Taut	0.46
While	0.49

Table 8. *Static vs. dynamic PRS schemes*

	Run-times		Hand-reductions per cycle	
	Dynamic	Static	Dynamic	Static
Fib	1.00	0.50	0.90	1.80
Queens	1.00	0.45	0.72	1.58

6.5 Modern processors

Modern microprocessors are the product of almost half a century of intensive engineering. Instruction pipelines with tens of stages have helped achieve clock frequencies in the region of 3–4 GHz. Techniques such as dynamic branch prediction, out of order execution and caching have enabled high utilisation of such deep pipelines.

The Reduceron, fully described in just 2,000 lines of Lava, exploits parallelism in a simpler way that arises very naturally in graph reduction: rather than process one word at a time, it processes several in parallel. It is *not* pipelined, so the sophisticated techniques needed to keep rapidly clocked pipelines busy are not needed.

6.6 The G-machine

In Peyton Jones (1987), template instantiation is presented as a ‘simple’ first step towards a more sophisticated approach to graph reduction based on the *G-machine*. So why is the Reduceron based on template instantiation and not the G-machine?

The G-machine approach aims to generate good code for *conventional hardware*, exploiting its strengths and avoiding its weaknesses. We base the Reduceron on template instantiation precisely because it does *not* make assumptions about the target hardware. The G-machine executes a sequential stream of fine-grained instructions, many of which could in fact be executed in parallel. The FPGA negates the assumption that such a sequential stream of instructions is necessary to avoid interpretive overhead.

6.7 Manipulating basic values

One aspect of reduction that the G-machine approach aims to optimise is the processing of basic values such as integers. In particular, avoiding construction of strictly needed primitive applications in memory can lead to large performance gains. For example, if a function body has the form `f ((+) x 1)` and `f` is strict then construction of `(+) x 1` on the heap can be avoided.

The Reduceron can also avoid construction of primitive applications to good effect (Section 5.3). However, it discovers suitable primitive applications *at run-time* and evaluates them *speculatively*. The Reduceron allows construction of `(+) x 1` to be avoided regardless of whether or not `f` is strict, but only if `x`, at run-time, takes the form `INT` *i*.

So the conditions under which construction of primitive applications can be avoided are quite different between the two approaches. As discussed in Section 5.3, strictness analysis can aid PRS. But strictness analysis alone, without some mechanism for reducing primitive redexes cheaply, is of little use to the Reduceron. PRS provides such a mechanism.

6.8 The SKIM machine

SKIM is a microcoded processor designed specifically to perform combinator reduction (Stoye, 1985). Stoye writes that ‘a combinator reducer coded on an 8-MHz 68000 goes at about one thirtieth of the speed of SKIM, and was considerably harder to write than SKIM’s microcode’.

One interesting aspect of SKIM is its use of *one-bit reference counts*. Stoye (1985) observes that such reference counts can be stored in the *pointer* to an application rather than in the application itself, making useful information about an application available without the expense of de-referencing a pointer. A reference count bit indicates whether the pointer is a *unique application pointer* or *multiple application pointer*. This information is used to good effect in SKIM: space pointed to by a unique pointer is reused rather than discarded. On average about 70% of used cells are immediately reused.

SKIM’s successful use of reference-count bits partly motivated the development of the dynamic sharing analysis presented in Section 5.1. We have precisely specified the modifications needed to implement dynamic sharing analysis in a general graph-reduction machine. We also discuss two important details not mentioned by Stoye (1985): (1) The subtle case in which an update can cause a unique pointer to become non-unique; and (2) Invariant 3, an important key to understanding why the technique actually works. We use the results of the analysis not for storage reclamation (which would complicate the machinery for template instantiation), but for update avoidance.

6.9 Static versus dynamic analysis

Sharing analysis. Burn *et al.* (1998) discuss the possibility of avoiding updates by identifying non-shared applications, including trade-offs between static and dynamic sharing analysis. The authors write that dynamic sharing analysis has the advantage of greater precision but that 'in general we strongly suspect that the cost of dashing greatly outweighs the advantages of precision when compared to [static analysis]'. In the Reduceron, dynamic sharing analysis (dashing) has *no time cost*: It is implemented in combinatorial logic that is not on the Reduceron's critical path. It is precise and simple to implement, requiring only minor modifications to three of the Reduceron's reduction rules.

Static analysis for primitive redex speculation. We have noted two important features of the Reduceron design: (1) By using wide parallel memories templates can be instantiated in a single clock-cycle. The size of templates is bounded, and if necessary templates are split, to make this single-cycle instantiation possible. (2) Primitive redexes in instances of templates are detected dynamically, and evaluated during instantiation – primitive redex speculation.

The cooperation of these two features is problematic. As templates must be determined statically at compile-time, but primitive speculation depends on a dynamic test, sizes of templates containing primitive applications must be reckoned in a way that allows for the worst case – *every primitive-redex test may fail.* So templates are split pessimistically and we do not gain the full benefit of PRS.

One way to avoid this problem is to perform a static analysis of the program. The goal of the analysis is to find all primitive applications whose every run-time instance can be guaranteed to be a redex.

We have some preliminary results from a trial of this approach for first-order programs. All Reduceron primitives have integer arguments. Our iterative analysis finds places where primitive speculation can sustain and exploit *valuable* data structures, in which integer components are already evaluated, without compromising non-strict semantics. If the pattern of valuable arguments to a function varies between applications, the definition of the function is cloned so that valuable cases can be exploited.

Benefits of static PRS include larger templates (since some primitive applications vanish) and simpler machinery for instantiation (since no PRS failures can occur). Preliminary results include significant gains for programs with an inner core of recurrent primitive computation. For example, Figure 8 shows the effect upon Fib and of Queens (naively de-functionalised). Besides better than 2× speed-ups, note the overall reduction rates well above one hand-reduction per cycle: the benefit of parallel primitive computation is clearly seen. Although Fib and Queens are examples for which static PRS is most advantageous, so far we have never observed it to perform less well than the dynamic scheme.

6.10 NORMA

Another special-purpose graph-reduction machine built in the 1980s is NORMA, the Normal Order Reduction MAchine (Scheevel, 1986). Like the Reduceron, NORMA used a wide instruction word (370 bits) and a wide data word containing a tag and two pointer-size components. NORMA also had an advanced register file allowing any number of data transfers between pairs of registers at the same time. In terms of performance, NORMA was capable of around 250,000 reductions per second. It had a clock frequency of around 5 MHz, giving a reduction rate of 0.125 reductions per cycle. NORMA was based on Turner's combinators, so a NORMA reduction is more fine-grained than a Reduceron reduction.

Unfortunately, at that time, building custom machines was a slow and expensive process, and any performance benefit obtained was wiped out by the next advance in stock hardware. In contrast, the Reduceron implementation is extremely cheap, thanks to FPGAs, and FPGAs are now an advancing technology in their own right.

6.11 The big word machine

A prototype machine similar in spirit to the Reduceron is Augustsson's Big Word Machine (BWM) (Augustsson, 1992). The BWM is a graph-reduction machine with a wide word size, four pointers long, allowing wide applications to be quickly built on, and fetched from, the heap. Augustsson likens the BWM to a very long instruction word (VLIW) machine (Hennessy & Patterson, 1992), designed for functional languages rather than scientific computing. Like the Reduceron, the BWM has a crossbar switch attached to the stack allowing complex rearrangements to be done in a single clock-cycle. The BWM also uses the Scott encoding to implement case expressions and constructors. Unlike the Reduceron, the BWM works on an explicit instruction stream rather than by template instantiation. The BWM was never actually built. Some simulations were performed but Augustsson writes: 'The absolute performance of the machine is hard to determine at this point'.

6.12 Ward's work

Ward (2000) explores the possibility of implementing a lazy functional language on FPGA by compilation to the hardware description language Handel-C. Consider the three main steps performed by a template instantiator running on a standard PC:

1. Fetch CPU instructions from memory, which, when executed,
2. read the function body from template memory and
3. construct an instance of the function body in graph memory.

The G-machine eliminates Step 2 by compiling each function to an instruction stream that constructs the body directly without having to traverse template memory. The Reduceron eliminates Step 1 by reading directly from template memory without having to execute any instructions. Ward (2000) proposes, in essence, to eliminate both Steps 1 and 2 by compiling functions into FPGA logic that directly instantiates the function body in graph memory. A potential attraction of this approach is that different functions can be executed at the same time. However, since every function definition requires its own connection to a single graph memory, an arbiter is needed to serialise memory accesses. And as the number of functions in a program grows, so too does the number of sites from which graph memory is accessed. It might be possible to split graph memory into independent units that can be accessed in parallel, but doing this effectively seems challenging. Ward's work did not progress beyond preliminary experiments.

6.13 The PilGRIM

A much more recent paper describes work on a processor design called *PilGRIM* for executing lazy functional languages (Boeijink *et al.*, 2011). The PilGRIM authors aim to explore a design that differs in two main ways from the Reduceron: (1) Use of an assembly-level instruction-set as opposed to template instantiation; and (2) use of hardware pipelining techniques to permit high clock frequencies. The authors write that 'with the extensive use of pipelining the PilGRIM targets a high clock frequency, however this comes with a great increase in the complexity of the design'.

In Boeijink *et al.* (2011), the PilGRIM instruction set and a compiler from a simple core functional language are presented. An actual synthesisable hardware description of the PilGRIM has not yet been produced; however, an instruction set simulator has been developed to gauge performance.

The performance of the PilGRIM is measured by counting the number of instructions needed to execute each program in the Reduceron benchmark suite. Across all the benchmarks, the PilGRIM instruction counts are very close to the Reduceron clock-cycle counts. So the performance of the PilGRIM would be similar to the Reduceron if it executed one instruction per cycle and clocked at 100 MHz. However, the PilGRIM authors are aiming for hardware which clocks at 1 GHz and executes one instruction every two clock-cycles on average. At the time of writing, to achieve such high clock frequencies would not be possible on an FPGA, so the PilGRIM authors are looking beyond FPGA-based hardware designs.

6.14 The statically allocated functional language

A quite different approach to running functional programs on FPGAs is taken in Mycroft and Sharp (2000). This paper presents the Statically Allocated Functional Language (SAFL), and shows how it can be compiled to FPGA. "Statically allocated" means that functions are restricted to be first-order, strict and tail-recursive (or not recursive at all). The motivation for compiling functional programs to hardware is that they contain a lot of fine-grained implicit parallelism. To illustrate, consider the following SAFL program:

```
fun mult(x, y, acc) =
  if (x = 0 or y = 0) then acc
  else mult(x<<1, y>>1, if y[0] then acc+x else acc)
```

Here `x` and `y` are the inputs to be multiplied, `acc` is an accumulator and the return value is the result of the multiplication. Each parameter in the recursive call to `mult` is evaluated in parallel and so too is each disjunct in the `or` expression.

The static-allocation restriction is quite prohibitive for general-purpose programming, and rules out a large class of idiomatic functional programs. However, pure SAFL extended with channels and mutable arrays is sufficient to express a Data Encryption Standard encryption/decryption circuit (Sharp, 2002). Frankau (2004) extends SAFL with similar features, but in a purely functional manner through lazy lists and linear-type arrays. Both static allocation and purity are enforced using a linear type system.

7 Conclusions and future work

Considering their relatively low clocking frequencies, FPGA applications must exploit significant parallelism to achieve high performance. In the context of sequential graph-reduction, we have taken this idea to its natural limit: Each reduction rule is performed in one clock-cycle. Furthermore, an FPGA circuit synthesised from our design achieves a respectable clock frequency compared to similar FPGA circuits for the same device. It is therefore quite hard to see how the Reduceron's reduction rules could be performed more quickly on an FPGA.

On the other hand, there is a lot of scope to reduce the *number* of reductions performed in a given program run. To this end, update avoidance and speculative evaluation of primitive redexes are both effective, making use of simple and precise dynamic analyses. These dynamic analyses would have a prohibitive run-time overhead in compiler-generated code running on a conventional machine, but have no such overhead in the Reduceron.

Our FPGA implementation of the Reduceron is on average 4–5 times slower than conventionally-compiled code running on a desktop PC (which has a 30 times higher clock frequency), but is 5–10 times faster than conventionally-compiled code running on a standard RISC processor on the same FPGA.

Future work. The main limitation of the current Reduceron implementation is the small amount of heap memory it provides. Could the heap be implemented using a larger, off-chip memory unit? We believe it could, without loss of performance, and without significant modification to the existing design. Two possible options are (1) the use of low-latency memory technologies such as RLDRAM, ZBT RAM or QDR SRAM, commonly used by FPGA applications that require access to large amounts of memory; and (2) the use of buffers or caches, implemented using on-chip block RAM.

Functional languages offer much scope for *parallel* evaluation of expressions. On conventional architectures there is a high cost for operations such as locking and releasing expressions under evaluation, so the benefits of parallel evaluation are offset by significant communication overheads. It would be interesting to see if special-purpose hardware could be used to overcome such overheads. Multiple Reducerons could be synthesised to FPGA, coordinated for parallel graph reduction.

One of the main features of FPGAs that we are not exploiting is that they can be configured on a *per-program* basis. One option would be to allow programmers to express, *as part of their program*, custom FPGA logic that accelerates execution of that program. Such logic would act as a *co-processor* to the Reduceron, and could itself be suitably described in the functional source language.

The future development and competitiveness of special-purpose processors for graph reduction remains questionable. But within a few years, just as plug-in GPU cards are already used for high-performance graphics, we would like to see FPU cards for high-performance applications of functional languages. We hope our work on the Reduceron makes a small advance in that direction.

Acknowledgments

This work was supported by the Engineering and Physical Sciences Research Council of the UK under grant EP/G011052/1. Thanks to Xilinx for donating the FPGA used in this work, and to Arjan Boeijink, Satnam Singh, Gabor Greif and anonymous ICFP and JFP reviewers for their helpful suggestions.

References

Augustsson, L. (1992) BWM: A concrete machine for graph reduction. In *Proceedings of the 1991 Glasgow Workshop on Functional Programming*. New York: Springer, pp. 36–50.

Boeijink, A., Holzenspies, P. K. F. & Kuper, J. (2011) Introducing the PilGRIM: A processor for executing lazy functional languages. In *Implementation and Application of Functional Languages (IFL 2010, Revised Selected Papers)*, Jurriaan Hage & Marco T. Morazán (eds), LNCS 6647. Berlin, Germany: Springer-Verlag, pp. 54–71.

Burn, G. L., Peyton Jones, S. L. & Robson J. D. (1988) The Spineless g-machine. In *Proceedings of the 1988 Conference on Lisp and Functional Programming*. New York: ACM, pp. 244–258.

Dijkstra, E. W. (1980) A mild variant of Combinatory Logic [online]. EWD735. Available at: `http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD735.PDF`. Accessed 26 June 2012.

Fasel, J. H. & Keller, R. M. (eds.) (1987) *Graph Reduction, Proceedings of a Workshop*, LNCS 279. New York: Springer.

Flanagan, C., Sabry, A., Duba, B. F. & Felleisen, M. (1993) The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI '93)*. New York: ACM, pp. 237–247.

Frankau, S. (2004) *Hardware Synthesis from a Stream-Processing Functional Language*. PhD Thesis, University of Cambridge, Cambridge, UK.

Gill, A. & Hutton, G. (2009) The worker/wrapper transformation. *J. Funct. Program.* **18**(2), 227–251.

Hennessy, J. & Patterson, D. (1992) *Computer Architecture; A Quantitative Approach*. Waltham, Massachusetts: Morgan Kaufmann.

Hutton, G. (2002) The countdown problem. *J. Funct. Program.* **12**(6), 609–616 (Cambridge University Press).

Jansen, J. M., Koopman, P. & Plasmeijer, R. (2007) Efficient Interpretation by transforming data types and patterns to functions. *Trends Funct. Program.* **7**, 157–172, Intellect.

Jones, R. & Lins, R. (1996) *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. San Francisco, CA: Wiley.

Kennaway, R. & Sleep, R. (1988) Director strings as combinators. *ACM Trans. Program. Lang. Syst.* **10**(4), 602–626.

Longbottom, R. (November 2009) Dhrystone Benchmark Results On PCs [online]. Available at: `http://www.roylongbottom.org.uk/dhrystone\%20results.htm`. Accessed 26 June 2012.

Mycroft, A. & Sharp, R. (2000) A Statically allocated parallel functional language. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*. New York: Springer, pp. 37–28.

Naylor, M. (2009a) An algorithm for arity-reduction, Reduceron memo 12 [online]. Available at: `http://www.cs.york.ac.uk/fp/reduceron/memos/Memo12.lhs`). Accessed 26 June 2012.

Naylor, M. (2009b) Design of the Octostack, Reduceron memo 27 [online]. Available at: `http://www.cs.york.ac.uk/fp/reduceron/memos/Memo27.lhs`.

Naylor, M. & Runciman, C. (2008) The Reduceron: Widening the von Neumann bottleneck for graph reduction using an FPGA. In *Implementation and Application of Functional Languages (IFL 2007, Revised Selected Papers)*, LNCS 5083. Berlin, Germany: Springer, pp. 129–146.

Naylor, M. & Runciman, C. (2010) The Reduceron reconfigured. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. New York: ACM, pp. 75–86.

Naylor, M., Runciman, C. & Reich, J. (2009) Reduceron home page [online]. Available at: `http://www.cs.york.ac.uk/fp/reduceron/`. Accessed 26 June 2012.

Nielson, H. R. & Nielson, F. 2007 *Semantics with Applications: An Appetizer*. New York: Springer.

Peyton Jones, S. L. (1987) *The Implementation of Functional Programming Languages*. Upper Saddle River, NJ: Prentice Hall.

Scheevel, M. (1986) NORMA: A graph reduction processor. In *Proceedings of the 1986 Conference on LISP and Functional Programming*. New York: ACM, pp. 212–219.

Scott, D. (1968) *A System of Functional Abstraction*. Notes of lectures delivered at University of California, Berkeley in 1962/1963. Stanford, CA: Stanford University.

Sharp, R. (2002) *Higher-Level Hardware Synthesis*. PhD Thesis, University of Cambridge, Cambridge, UK.

Stoye, W. (1985) *The Implementation of Functional Languages Using Custom Hardware*. PhD Thesis, University of Cambridge, Cambridge, UK.

Turner, D. A. (1979) A New implementation technique for applicative languages. *Softw. Pract. Exp.* **9**(1), 31–49.

Ward, M. (2000) *Supercombinator Soft Machines*. M Eng. Project Dissertation, University of York, York, North Yorkshire, UK.

Weicker, R. (1984) Dhrystone: A synthetic systems programming benchmark. *Commun. ACM* **27**(10), 1013–1030.

Xilinx (April 2009) MicroBlaze Soft Processor v7.20 [online]. . Available at: `http://www.xilinx.com/tools/microblaze.htm`. Accessed 26 June 2012.