Using Projection Analysis in Compiling Lazy Functional Programs[1]

G L Burn

Department of Computing, Imperial College, 180 Queen's Gate,
London SW7 2BZ. United Kingdom.

Electronic mail: glb@doc.ic.ac.uk

[1] This work was partially completed whilst the author was working for the GEC Hirst Research Centre, East Lane, Wembley, Middlesex HA9 7PP, UK, and so was partially funded by ESPRIT Project 415: Parallel Architectures and Languages for AIP – A VLSI-Directed Approach.

Abstract

Projection analysis is a technique for finding out information about lazy functional programs. We show how the information obtained from this analysis can be used to speed up sequential implementations, and introduce parallelism into parallel implementations. The underlying evaluation model is evaluation transformers, where the amount of evaluation that is allowed of an argument in a function application depends on the amount of evaluation allowed of the application. We prove that the transformed programs preserve the semantics of the original programs.

Compilation rules, which encode the information from the analysis, are given for sequential and parallel machines.

1 Introduction

A number of analyses have been developed which find out information about programs. The methods that have been developed fall broadly into two classes, forwards analyses such as those based on the ideas of abstract interpretation (e.g. [9, 18, 19, 7, 17, 12, 4, 20]), and backward analyses such as those based on projections (e.g. [22]), the work of Hall ([11]), and inverse image analysis ([10]).

The analysis techniques have mostly been applied to finding out information about the definedness of functions. This information has then been used to make more efficient implementations of functional programs. For example, the information can be used to compile code which builds fewer closures when executing a program on a sequential machine. On a parallel machine, it has been used to indicate which parts of a functional program can be evaluated in parallel.

The idea of an evaluation model for lazy functional programs, where the amount of evaluation allowed of an argument expression depends on the amount of evaluation allowed of the function application, was first presented in [4], and reported in [3]. It was called the *evaluation transformer model* because the amount of evaluation allowed of a function application is mapped to the amount of evaluation allowed of an argument expression. Abstract interpretation was used to find the information required by the model. In the same proceedings as [3], Wadler and Hughes presented their work on projection analysis [22]. The results of the analysis had strong resemblances to the information that was required for the parallel model we had developed. It was difficult to compare the two pieces of work however, for two reasons. Firstly, the evaluation transformer model of [3] was based on information obtained from abstract interpretation, whilst the work of [22] used projections. We addressed this issue in [6], proving a relationship between abstract interpretation and projection analysis. Secondly, the definitions and proofs of [4] were couched in terms of developing a parallel implementation, although the techniques were applicable to reasoning about sequential implementations, whilst the (informal) pragmatics of [22] were for a sequential implementation. We address this second issue, of using the information in an implementation, in this paper.

We will restrict our attention to the information available from projections. Whilst no satisfactory projection analysis has been developed for higher-order functions as yet, they do seem to be able to capture more types of information than abstract interpretation for first-order functions – see [6] for a discussion of the relationship between the two techniques. Projections also give us one extra piece of information. Sometimes they are able to say something about some functions that do not necessarily have to evaluate their arguments, whereas abstract interpretation does not appear to be able to give any information about a function when it cannot determine that an argument expression needs to be evaluated.

Specifically, this paper makes the following contributions:

- Projections capture the notion of being able to evaluate an expression to a certain extent (e.g. to weak head normal form (WHNF), or the structure of a data object). We formalise this by introducing the operational notion of an *evaluator*. An evaluator can be implemented in a number of different ways, as long as some semantic constraints are met. (See Section 3.)

- Evaluators form the link between the denotational and operational semantics of a program. We are able to use the denotational information from projection analysis to allow a change in the operational behaviour of a program, so that the denotational semantics of the program is preserved. A number of theorems show how the information can be used in this way, including the use of the information even when a closure has to be created in the heap. For simplicity of presentation, all theorems will be proved for functions taking one argument. (See Sections 4.1 and 4.2.)

- A corollary of the theorems is that a function does not have to be strict in order to be able to change the evaluation order of the program. (See Section 4.1.)

- In a graph reduction implementation, using the information from Sections 4.1 and 4.2 in a naïve way

could cause errors when closures for expressions are shared. We add some extra restrictions in order to make the model safe in a graph reduction framework. (See Section 4.3.)

- We provide compilation rules which show how the information from projections can be used in both sequential and parallel implementations. This shows how the compilation rules for the sequential and parallel machines are very similar, essentially choosing a different way of implementing the evaluator, and evaluating arguments 'strictly' or in parallel. We thus clarify some of the issues raised in the discussion of [22] and [3]. A version of the code is produced for each evaluator supported by the machine. (See Section 5.)

- The implementor can choose whether to have the version of a function chosen at compile-time or run-time. The latter gives more opportunities for optimisation, at the expense of some extra state on each node, and having to check the state when the evaluation of an expression begins. (See Section 5.3.)

We conclude the paper with some discussion of the proofs in the paper and give some pointers to literature describing implementations of the model on parallel machines.

As a matter of notation, throughout the paper we will denote pieces of program text by writing them in type-writer font. For example, `E` represents an arbitrary expression from a program. The semantics of an expression `E` will be written in italics, E. Sometimes, to emphasise that we are referring to the semantics of a piece of program `E`, we will write $\mathcal{M}$ $[\![\texttt{E}]\!]$.

2 Some Preliminary Definitions, Facts and Lemmata

The body of this paper relies on a number of definitions, facts and lemmata, given in this section. Some of the definitions are standard; the rest are taken from one of [22, 4, 6].

As an example application of projection analysis, we will use projections for lists. The relevant projections are defined in Section 2.2.

2.1 General Definitions and Results

For each type, a domain which contains all of the elements of that type must be given. If σ is a type, then D_σ is the domain which is the standard interpretation of the type. This is the domain in which the normal semantics of an expression of that type is given a value.

In order to capture the concept of a function needing to evaluate an argument (to a certain extent), Wadler and Hughes introduced a new domain element called $\lightning$, which is less defined than $\bot$. Adding a new domain element in this way can be formalised using domain *lifting*, where we refer to the bottom element of the lifted domain rather than $\lightning$.

Definition 2.1

If σ is a type, then we denote the lifted type by $lift\ \sigma$.

$lift$ is a functor over domains, which means that it can be applied to the objects of the category of domains (domains) and the arrows in the category of domains (continuous maps). If D_σ is the domain for the type σ, then $lift\ D_\sigma$ is the domain for the type $lift\ \sigma$. If D is a domain, then the elements of $lift\ D$ are $\{\bot_{lift\ D}\} \bigcup \{<0, d> | d \in D\}$. The ordering on the domain is defined by :

$$\begin{array}{lll} \bot_{lift\ D} & \sqsubseteq\ d & \forall d \in (lift\ D) \\ <0,\ d> & \sqsubseteq <0,\ d'> & \text{iff } d \sqsubseteq d' \end{array}$$

If $f : \sigma \rightarrow \tau$, then $lift\ f : lift\ \sigma \rightarrow lift\ \tau$, where $lift\ f$ is defined by :

$$\begin{array}{ll} (lift\ f)\ \bot_{lift\ D_\sigma} & = \bot_{lift\ D_\tau} \\ (lift\ f)\ <0, d> & =<0,\ f\ d> \end{array}$$

The inverse of $lift$ is called *drop*, and is defined in the obvious way.

Definition 2.2

- A function α is a *projection* if it is idempotent and less than the identity, that is, $\alpha \circ \alpha = \alpha$, and $\alpha \sqsubseteq ID$, where ID is the identity function.
- The most defined projection is ID, defined by

  $$ID\ u = u$$

  for all u, and the least projection is BOT, defined by

  $$BOT\ u = \bot$$

  for all u.
- A projection $\alpha : lift\ \sigma \rightarrow lift\ \sigma$ is *lift-strict* if

  $$\alpha\ <0,\ \bot_{D_\sigma}>\ = \bot_{lift\ D_\sigma}.$$[2]

[2] In [22], such projections are simply called *strict*. However, we feel that this is a confusing use of terminology, especially when projections are recast into a world using lifted domains.

The greatest lift-strict projection is STR, defined by

$$\begin{array}{lcl} STR\ \bot_{lift\ D_\sigma} & = & \bot_{lift\ D_\sigma} \\ STR\ <0,\ \bot_{D_\sigma}> & = & \bot_{lift\ D_\sigma} \\ STR\ <0,\ d> & = & <0,\ d> \\ & & \text{if } d \neq \bot_{D_\sigma} \end{array}$$

- For any projection $\alpha : \sigma \rightarrow \sigma$, the corresponding lift-strict projection is $STR \circ (lift\ \alpha)$. The lift-strict projection corresponding to a projection α will be denoted by α^S,[3] that is,

$$\alpha^S = STR \circ (lift\ \alpha).$$

- A projection $\alpha : \sigma \rightarrow \sigma$ is a *smash* projection if $\forall u \in D_\sigma$ either $\alpha\ u = \bot_{D_\sigma}$ or $\alpha\ u = u$.

The projection ABS is defined in [22] to model not needing to evaluate (part) of an argument expression.

Definition 2.3

$$\begin{array}{lcl} ABS\ \bot_{lift\ D_\sigma} & = & \bot_{lift\ D_\sigma} \\ ABS\ <0,\ d> & = & <0, \bot_{D_\sigma}> \end{array}$$

Note that $ABS = lift\ BOT$. We make an observation about its use in Section 4.3.

Definition 2.4

If α is a projection, f a function and $f \circ \alpha = f$, then we will say that f is α-strict in its argument. More generally, if

$$\begin{array}{l} \forall d_1, \ldots, d_n,\ \beta(f\ d_1\ \ldots\ d_n) \\ \quad = \beta\ (f\ d_1\ \ldots\ d_{i-1}\ (\alpha\ d_i)\ d_{i+1}\ \ldots\ d_n) \end{array}$$

then we will say that f is α-strict in its ith argument in a β-strict context, and write it $F_i : \beta \Rightarrow \alpha$. The *projection transformer* for the ith argument of the function f is the map which, given some projection β, returns the least projection α which satisfies the above equation. We will also denote this by F_i.

Lemma 2.5

$lift\ f : \beta^S \Rightarrow \alpha^S$ implies $f : \beta \Rightarrow \alpha$.

Proof

There are two cases to consider:

1. $\alpha\ u = \bot$: This follows from Lemma 4.1.
2. $\alpha\ u \neq \bot$:

$$\begin{array}{l} STR\ <0, \beta\ (f\ (\alpha\ u))> \\ \quad = (\beta^S \circ (lift\ f) \circ \alpha^S)\ <0, u> \\ \quad = (\beta^S \circ (lift\ f))\ <0, u> \\ \qquad \text{condition of Lemma} \\ \quad = STR\ <0, \beta\ (f\ \ u)> \end{array}$$

and so we may conclude that $\beta\ (f\ (\alpha\ u)) = \beta\ (f\ \ u)$.

□

Definition 2.6

By $\alpha \sqcap \beta$ we mean the largest projection δ such that $\delta \sqsubseteq \alpha$ and $\delta \sqsubseteq \beta$ [22, p.390].

Lemma 2.7

Suppose that α is a smash projection, and β is any other projection. Then

1. *$\alpha \circ \beta$ is a projection;*
2. *$\alpha \circ \beta = \alpha \sqcap \beta$; and*
3. *$\alpha \circ \beta$ is not necessarily a smash projection.*

Proof

We prove each of the statements separately.

1. $\alpha \circ \beta$ is a projection: $\alpha \circ \beta \sqsubseteq ID$ trivially since α and β are projections. Since α is a smash projection, there are two cases to consider in order to show that $\alpha \circ \beta$ is idempotent:

   (a) $\alpha\ (\beta\ u) = \bot$:

$$\begin{array}{ll} \alpha\ (\beta\ (\alpha\ (\beta\ u))) & = \alpha\ (\beta\ \bot) \\ & = \bot \\ & = \alpha\ (\beta\ u) \end{array}$$

   (b) $\alpha\ (\beta\ u) = \beta\ u$:

$$\begin{array}{ll} \alpha\ (\beta\ (\alpha\ (\beta\ u))) & = \alpha\ (\beta\ (\beta\ u)) \\ & = \alpha\ (\beta\ u) \end{array}$$

2. Let δ be any projection such that $\delta \sqsubseteq \alpha$ and $\delta \sqsubseteq \beta$. Then for any u, $\delta\ u = \delta\ (\delta\ u) \sqsubseteq \alpha\ (\delta\ u) \sqsubseteq \alpha\ (\beta\ u)$. Since $\alpha \circ \beta \sqsubseteq \alpha$, $\alpha \circ \beta \sqsubseteq \beta$, $\alpha \circ \beta$ is a projection (part (1) of this Lemma), and by definition $\alpha \sqcap \beta$ is the greatest projection satisfying this property, the above calculation shows that $\alpha \circ \beta$ must be $\alpha \sqcap \beta$.

[3] This is different from the convention of [22], where lift-strict projections are primed.

3. If H_B and H are the projections defined in Figure 1, we can see that H_B is a smash projection and that $H_B \circ H$ is not a smash projection. For example, $(H_B \circ H)\ (cons\ 1\ (cons\ \bot\ E)) = cons\ 1\ \bot$, which is less defined than $(cons\ 1\ (cons\ \bot\ E))$, but not $\bot_{D_{Alist}}$.

□

Note that the composition of two projections is not in general a projection. For example, $H \circ T$, where H and T are defined in Figure 1, is not a projection.

Lemma 2.8

If α is a smash projection, and β any projection, then $\alpha \circ \beta \sqsubseteq \beta \circ \alpha$.

Proof

There are two cases to consider:

1. $\alpha\ u = \bot$: Since α and β are projections, $\alpha\ (\beta\ u) \sqsubseteq \alpha\ u = \bot$, and for this case $\beta\ (\alpha\ u) = \beta\ \bot = \bot$.
2. $\alpha\ u = u$: Since α and β are projections, $\alpha\ (\beta\ u) \sqsubseteq \beta\ u = \beta\ (\alpha\ u)$.

□

2.2 An Application : Lists

We are interested in the five projections ID, T, H, H_B, and $T \circ H$, and their lift-strict versions. ID is the identity function, and T, H, and H_B are defined in Figure 1; T and H were originally defined in [22], and H_B in [6]. $T \circ H$ is a projection by Lemma 2.7. If A is some type, then we are interested in *Alists*, which are lists of elements of type A. By D_A^* we mean a finite or empty sequence of elements from D_A, and D_A^ω means an infinite sequence.

Both H and H_B capture notions of head-strictness. Intuitively, a function being H-strict means that, any part of the list which is evaluated also has to have the elements of the list evaluated, whilst being H_B-strict means that, if it needs to evaluate the list, then it needs to evaluate the head as well. The relationship between H and H_B was discussed in [6].

3 Evaluators and Safety: Linking Denotational and Operational Notions

Informally, the fact that a function satisfies the equation:

$$\beta \circ f \circ \alpha = \beta \circ f$$

$$T\ u = \begin{cases} \bot & if\ u \in \{D_A^* . \bot_{D_{Alist}}\} \bigcup \{D_A^\omega\} \\ u, & otherwise \end{cases}$$

$$\begin{array}{ll} H\ \bot_{D_{Alist}} & = \bot_{D_{Alist}} \\ H\ (cons\ \bot_{D_A}\ e) & = \bot_{D_{Alist}} \\ H\ (cons\ u\ e) & = cons\ u\ (H\ e),\ u \neq \bot_{D_A} \\ H\ nil & = nil \end{array}$$

$$\begin{array}{ll} H_B\ \bot_{D_{Alist}} & = \bot_{D_{Alist}} \\ H_B\ (cons\ \bot_{D_A}\ e) & = \bot_{D_{Alist}} \\ H_B\ (cons\ u\ e) & = cons\ u\ e,\ u \neq \bot_{D_A} \\ H_B\ nil & = nil \end{array}$$

Figure 1: Some Projections for Lists

says that, whenever an application of f can be replaced by β applied to the application, then we can replace the argument to f by α applied to the argument. But what do we mean by *replace*? Consider the function `before`, defined in Figure 2, which returns the part of

```
before []       = []
before (x:xs)   = if x=0 then []
                         else x:before xs
```

Figure 2: The function `before`.

its argument list which precedes the first `0`, if one exists. It has the following property:

$$ID \circ before \circ H = ID \circ before$$

To replace the argument of `before` by H applied to it clearly cannot mean to *evaluate* H applied to the argument expression. For example, `(before [0,⊥])` is `[]`, but evaluating $H\ [0, \bot]$ will not terminate. Therefore, we have developed the notion of an *evaluator*, which captures the behaviour of a projection in an operational way.

Given some projection α, let us try to define an evaluator which can be said to *correspond* to the projection. At an abstract level, we can think of the implementation of a lazy functional language as a term-rewriting system. In a term rewriting system we have to define a *reduction strategy*, which specifies the order in which redexes are chosen for reduction. Lazy evaluation is a reduction strategy which chooses the left-most outermost redex, and evaluates expressions to WHNF. A projection

could be modelled by an alternative reduction strategy, which stopped evaluation at some other 'normal form'. What properties should the evaluator have? Our intuition about projections is that if a projection maps a value to $\perp$, and it is safe to replace the value by the projection applied to the value, then this corresponds to being allowed to initiate an infinite computation when evaluating the expression. Let us capture this in a first attempt at defining an evaluator corresponding to a projection.

Definition 3.1

> An *evaluator* corresponding to a projection α is a reduction strategy such that, given any expression `E` of the appropriate type, it initiates an infinite computation when evaluating `E` if and only if $\alpha\ (\mathcal{M}\ [\![\texttt{E}]\!]) = \perp$[4].

Before we discuss this definition, we have to define what we mean by an 'infinite' computation.

Definition 3.2

> An *infinite* computation is one which never terminates.

Note that an infinite computation is not the same as $\perp$; as well as failing to terminate, it may compute any finite approximation to the result.

The main problem with the definition is that it does not capture the full meaning of a projection. For example, an evaluator corresponding to H is identical to one corresponding to H_B. The difference between the two projections is that H may change the value of some object to something less defined than itself, which is not bottom. With H, the evaluator should try to evaluate the expression to a `cons` cell with an evaluated head, and leave a closure in the heap for the tail so that its semantics is the same as H applied to it. Our definition of evaluators does not capture this because each reduction step preserves the semantics of the program. We therefore make a second attempt at a definition.

Definition 3.3

> An *evaluator* corresponding to a projection α is a rewriting strategy such that for all expressions `E` of the appropriate type:
>
> 1. it initiates an infinite computation when evaluating `E` if and only if $\alpha\ (\mathcal{M}\ [\![\texttt{E}]\!]) = \perp$;
> 2. if the evaluation of an expression `E` terminates with expression `E'`, then we have that
>
> $$\mathcal{M}\ [\![\texttt{E}']\!] = \alpha\ (\mathcal{M}\ [\![\texttt{E}]\!]).$$

Notice that we have changed from calling it a reduction strategy to calling it a rewriting strategy, because we allow the semantics of an expression to be altered[5]. Also notice that the definition does not force the implementor to choose any particular rewriting strategy to implement an evaluator. For example, an evaluator corresponding to T may evaluate all of the tails of the argument list before returning any information. In this case, when T returns $\perp$, the infinite computation initiated by the evaluator never gives any information. Another way of implementing T is to define a process which evaluates the argument to WHNF, and then creates a process which will do the same on the tail of the list. This particular evaluator will produce whatever part of the list structure is defined, and so has a different sort of infinite behaviour. However, all evaluators corresponding to a projection have the property that when evaluating an expression, they all fail to terminate, or they terminate with expressions which all have the same semantics. We therefore feel excused in referring to *the* evaluator corresponding with the projection α, and write it as ξ_α.

The whole point of doing the projection analysis is so that we can find out when we can change the rewriting strategy from lazy evaluation to something more efficient. We must formalise what it means to be safe to change the evaluation strategy of a program.

Definition 3.4

> It is *safe* to change the evaluation strategy of a program if the answer from the resulting evaluation gives the same answer as the denotational semantics of the original program, and doing so only introduces infinite computations if the semantics of the program is $\perp$.

4 Projections, Evaluators and Operational Semantics

In this section we show how to take the information from projection analysis and use it to tell us which evaluator can be used to evaluate an expression, and when. When

[4] This actually forces a canonical form for expressions which have been evaluated with an evaluator corresponding to a particular projection. For example, according to this definition, an evaluator corresponding to ID evaluates an expression to WHNF. To see this, note that it must fail to terminate if and only if the semantics of an expression is $\perp$. Therefore, evaluation of (`cons` $\perp$ $\perp$) must terminate, and so the evaluator cannot evaluate arguments to `cons`, and it must evaluate at least to WHNF because it has to fail to terminate if the semantics of the expression is $\perp$.

[5] Our theorems later on will ensure that we only allow the semantics of an expression to be altered if it does not change the meaning of the entire program.

a function satisfies certain conditions, we will be able to change the evaluation order, whilst at other times we will have to create a closure for the expression, but we will be able to mark it so that, if the expression is ever evaluated, then it can be evaluated with a particular evaluator. If an implementation uses graph reduction, then a naïve use of the theorems in Sections 4.1 and 4.2 can cause errors. We show how to fix the situation in Section 4.3.

4.1 Changing the Evaluation Order

Wadler and Hughes introduced the idea of a lift-strict projection in [22] in order to capture the concept of a function *needing* to evaluate its argument. With the definitions we have given, we are able to prove a theorem which shows how lift-strict projections can be used to allow the evaluation order of a program to be changed. It is similar to Theorem 4.2.1.1 of [4]. But first a Lemma.

Lemma 4.1

Suppose that $lift\ f : \beta^S \Rightarrow \alpha^S$, *where* $f : \sigma \rightarrow \tau$. *Then for all* u *such that* $\alpha\ u = \bot_{D_\sigma}$, $\beta\ (f\ u) = \bot_{D_\tau}$.

Proof

Let u be any value such that $\alpha\ u = \bot_{D_\sigma}$.

$$
\begin{array}{ll}
STR\ <0, \beta\ (f\ u)> & \\
= \beta^S\ <0, f\ u> & \text{by definition} \\
= (\beta^S \circ (lift\ f))\ <0, u> & \text{by definition} \\
= (\beta^S \circ (lift\ f) \circ \alpha^S)\ <0, u> & \\
\quad \text{since } lift\ f : \beta^S \Rightarrow \alpha^S & \\
= \beta^S\ ((lift\ f)\ \bot_{lift\ D_\sigma}) & \text{since } \alpha\ u = \bot_{D_\sigma} \\
= \beta^S\ \bot_{lift\ D_\tau} & \\
= \bot_{lift\ D_\tau} &
\end{array}
$$

which means that $\beta\ (f\ u) = \bot_{D_\tau}$ by the definition of STR.

□

Theorem 4.2

Suppose that `f` $: \sigma \rightarrow \tau$, *that* $\alpha : \sigma \rightarrow \sigma$ *and* $\beta : \tau \rightarrow \tau$ *are projections, and that* $(lift\ f) : \beta^S \Rightarrow \alpha^S$. *Then it is safe to evaluate the argument to* `f` *with evaluator* ξ_α *when it is safe to evaluate an application of* `f` *with evaluator* ξ_β.

Proof

Suppose that in an application of `f` to an expression `E`, it is safe to evaluate the application with the evaluator ξ_β. There are two cases to consider:

1. The evaluation of the expression `E` with evaluator ξ_α does not initiate an infinite computation: This is safe because $\beta\ (f\ (\alpha\ (\mathcal{M}\ [\![\texttt{E}]\!])))) = \beta\ (f\ (\mathcal{M}\ [\![\texttt{E}]\!]))$ (Lemma 2.5), and it is safe to evaluate the application with ξ_β, and so the meaning of the program is not changed.
2. The evaluation of the expression `E` with evaluator ξ_α initiates an infinite computation: In this case, by the definition of ξ_α, we are assured that $\alpha\ (\mathcal{M}\ [\![\texttt{E}]\!]) = \bot_{D_\sigma}$, and hence that $\beta\ (f\ (\mathcal{M}\ [\![\texttt{E}]\!])) = \bot_{D_\tau}$, by Lemma 4.1, above. This means that the evaluation of the application of `f` with ξ_β will initiate an infinite computation. Since it was safe to evaluate the application with ξ_β, the semantics of the original program must have been $\bot$, and hence it was safe to initiate an infinite computation when evaluating the argument expression.

□

It may seem unnecessary to use lift-strict projections in order to show that the evaluation order can be changed. However, the following theorem shows that it is not sufficient just to know that $f : \beta \Rightarrow \alpha$.

Theorem 4.3

Suppose that `f` $: \sigma \rightarrow \tau$, *that* $\alpha : \sigma \rightarrow \sigma$ *and* $\beta : \tau \rightarrow \tau$ *are projections, and that* $f : \beta \Rightarrow \alpha$. *Then it is not necessarily safe to evaluate the argument to* `f` *with evaluator* ξ_α *when it is safe to evaluate an application of* `f` *with evaluator* ξ_β.

Proof

Consider the function

```
f xs = []
```

For any projections α and β, we have that $(\beta \circ f \circ \alpha) = \beta \circ f$, but it is clearly unsafe to do any evaluation of the argument to `f`.

□

Theorem 4.2 gave a *sufficient* condition for allowing the evaluation order to be changed, and Theorem 4.3 showed that a function being α-strict in a β-strict context is not. The next theorem shows that, from the fact that f is strict and $f : \beta \Rightarrow \alpha$, we can conclude that the conditions for Theorem 4.2 hold. Following that, we give a theorem which shows that it is possible for a function to satisfy the conditions of Theorem 4.2 without it being strict. This says that:

Strictness is a sufficient *but not a* necessary *condition for the evaluation order to be changed, contrary to the folklore about program analysis techniques.*

This result is implicit in Theorems 4.2.1.1 and 4.2.2.1 of [4].

Theorem 4.4

$f : \beta \Rightarrow \alpha$ and f strict implies $(lift\ f) : \beta^S \Rightarrow \alpha^S$.

Proof

There are three cases to consider.

1.
$$\begin{aligned}&(\beta^S \circ (lift\ f) \circ \alpha^S)\ \perp_{lift\ D_\sigma}\\ &= \perp_{lift\ D_\tau}\\ &= (\beta^S \circ (lift\ f))\ \perp_{lift\ D_\sigma}\end{aligned}$$

2.
$$\begin{aligned}&(\beta^S \circ (lift\ f) \circ \alpha^S)\ <0, \perp_{D_\sigma}>\\ &= (\beta^S \circ (lift\ f))\ \perp_{lift\ D_\sigma}\\ &= \beta^S\ \perp_{lift\ D_\tau}\\ &= \beta^S <0, \perp_{D_\tau}>\\ &= \beta^S <0, f\ \perp_{D_\sigma}>\\ &\quad \text{since } f \text{ is strict}\\ &= (\beta^S \circ (lift\ f))\ <0, \perp_{D_\sigma}>\end{aligned}$$

3. Let $u \neq \perp_{D_\sigma}$.
$$\begin{aligned}&(\beta^S \circ (lift\ f) \circ \alpha^S)\ <0, u>\\ &= (\beta^S \circ (lift\ f))\ <0, \alpha\ u>\\ &= \beta^S <0, f\ (\alpha\ u)>\\ &= STR\ <0, \beta\ (f\ (\alpha\ u))>\\ &= STR\ <0, \beta\ (f\ u)>\\ &\quad \text{since } f : \beta \Rightarrow \alpha\\ &= \beta^S\ <0, f\ u>\\ &= (\beta^S \circ (lift\ f))\ <0, u>\end{aligned}$$

□

Theorem 4.5

There exist functions $f : \beta \Rightarrow \alpha$ which are not strict, but which satisfy $(lift\ f) : \beta^S \Rightarrow \alpha^S$.

Proof

Consider the function `f` defined by :

```
f xs = append [9] xs
```

We have that $f : T \Rightarrow T$, $(lift\ f) : T^S \Rightarrow T^S$ and f is not strict.

□

4.2 Creating Annotated Closures

The function

```
f x ys = if x=0 then [] else length ys
```

satisfies $F_2 : ID \Rightarrow T$ but not $F_2 : ID^S \Rightarrow T^S$[6]. Therefore, we have to build a closure for any argument to `f`. However, it is clear that if an argument is ever evaluated, then it is safe to evaluate it with ξ_T. The following theorem shows that this can be done in general.

Theorem 4.6

Suppose that `f` $: \sigma \rightarrow \tau$, that $\alpha : \sigma \rightarrow \sigma$ and $\beta : \tau \rightarrow \tau$ are projections, that $f : \beta \Rightarrow \alpha$, and that ξ_β is a safe evaluator for some application of `f`. Then, if the argument is ever evaluated, then it is safe to evaluate the argument to `f` with evaluator ξ_α.

Proof

Suppose that `f` is being applied to some argument expression `E`, and that ξ_β is a safe evaluator for the application. Then it must be safe to replace the application by β (`f E`). Since $f : \beta \Rightarrow \alpha$, the expression `E` can be replaced by α `E`. Suppose that the expression α `E` is eventually evaluated. Then the evaluation will fail to terminate if and only if $\alpha\ (\mathcal{M}\ [\![\texttt{E}]\!]) = \perp_{D_\sigma}$, and if it terminates, the resulting expression will have semantics equal to $\alpha\ (\mathcal{M}\ [\![\texttt{E}]\!])$. This is exactly what happens if the expression `E` is evaluated with ξ_α.

□

When Theorem 4.2 says that an expression can be evaluated with ξ_α, or Theorem 4.6 says that a closure can be created for the argument such that, if it is ever evaluated, then it can be evaluated with ξ_α, we will say that the expression is *marked* with the evaluator ξ_α.

4.3 A Complication of Graph Reduction

Unfortunately, we have to be careful how we apply Theorems 4.2 and 4.6 in a graph reduction implementation. The theorems essentially assume that the function being applied has its own copy of the argument expression. However, in a graph reduction implementation, argument expressions are shared, and a naïve application of the two theorems can alter the semantics of a program. For example, consider the functions defined

[6] Please excuse the abuse of notation. Strictly speaking, this information is about the lifted functions, which only take one argument. We hope the intended meaning is clear, and will continue to abuse the notation throughout the rest of the paper.

in Figure 3. With these definitions, the application (`f [0,⊥]`) should return the singleton list `[2]`. The following projection information is relevant to our example:

$$CONS_2 : (T \circ H)^S \Rightarrow (T \circ H)^S$$
$$BEFORE_1 : (T \circ H)^S \Rightarrow H^S$$

Suppose the application `f [0,⊥]` is being evaluated with evaluator $\xi_{(T \circ H)}$. Then following the projection information through, and using Theorem 4.2, we see that the argument to `before` can be evaluated with ξ_H. Unfortunately, in a graph reduction implementation, this argument is a closure shared with the argument of `length`. If we evaluated the argument with ξ_H, the expression left will the same semantics as `[0,` $(H \ \bot)$`]`, that is `0:`$\bot$. When `length` is applied to this, it will fail to terminate. This is not safe because because the result of the application should have been `[2]`, and $\xi_{(T \circ H)}$ would not have initiated an infinite computation in evaluating this. The problem arises because ξ_H has changed the meaning of the expression which is the argument to `length`.

```
f xs = (length xs):(before xs)

before []       = []
before (x:xs)   = if x=0 then []
                         else x:before xs

length []       = 0
length (x:xs)   = 1 + length xs
```

Figure 3: An Example to Show the Problems of Graph Reduction

This example shows that we need to make a distinction between two types of function application in the source text of a program. The first is a function applied to another function application. Here Theorems 4.2 and 4.6 can be used because the projection information about the function is guaranteed to be true of its argument. The second case is when a function is applied to a formal parameter. Here the argument is a closure in the heap, which may be shared by other function applications, and the projection information can only be used if it does not interfere with the values of any other expressions which share the closure. Theorem 4.8 below shows how we can ensure this in general. The statement of the theorem makes use of the following definition.

Definition 4.7

Given any projection α, define *smash* α to be the least smash projection δ such that $\alpha \sqsubseteq \delta$.

For example, *smash* $H = H_B$. Smash projections are useful because their corresponding evaluators either fail to terminate or leave the semantics of the expression undisturbed.

Theorem 4.8

Suppose that it is safe to evaluate an application of `f` *with evaluator* ξ_β*, that the argument to* `f` *is marked with the evaluator* ξ_γ*, and that lift* $f : \beta^S \Rightarrow \alpha^S$*. Then it is safe to evaluate the argument expression with* $\xi_{(smash\ \alpha) \circ \gamma}$*.*

Proof

Let us call the argument expression `E`, and note that from Lemma 2.7, since (*smash* α) is a smash projection, we have that (*smash* α) $\circ\ \gamma$ is a projection, and so it makes sense to write $\xi_{(smash\ \alpha) \circ \gamma}$ The fact that `E` was marked with ξ_γ means that it is safe to evaluate the expression with ξ_γ, and so the semantics of the program are not altered if the expression is replaced with an expression which has meaning $\gamma\ (\mathcal{M}\ [\![\texttt{E}]\!])$. There are two cases to consider.

1. (*smash* α) $(\gamma\ (\mathcal{M}\ [\![\texttt{E}]\!])) = \gamma\ (\mathcal{M}\ [\![\texttt{E}]\!])$: Here evaluating the expression with $\xi_{(smash\ \alpha \circ \gamma)}$ is exactly the same as evaluating it with ξ_γ, and as it is safe to evaluate the expression with ξ_γ, it is safe to evaluate it with $\xi_{(smash\ \alpha \circ \gamma)}$.
2. (*smash* α) $(\gamma\ (\mathcal{M}\ [\![\texttt{E}]\!])) \neq \gamma\ (\mathcal{M}\ [\![\texttt{E}]\!])$: Since *smash* α is a smash projection, this means that (*smash* α) $(\gamma\ (\mathcal{M}\ [\![\texttt{E}]\!])) = \bot$, and so the evaluation fails to terminate. However, by Lemma 4.1, this means that $\beta\ (f\ (\gamma\ (\mathcal{M}\ [\![\texttt{E}]\!]))) = \bot$, and hence the evaluation of the application with ξ_β will fail to terminate. Since it is safe to evaluate the application with ξ_β, the semantics of the program must be $\bot$, and so it was safe to evaluate the argument expression with $\xi_{(smash\ \alpha) \circ \gamma}$.

□

There are two things to note about this theorem. Firstly, we cannot replace *smash* α with α for any projection α, nor are we allowed to just have that $f : \beta \Rightarrow \alpha$ and not $lift\ f : \beta^S \Rightarrow \alpha^S$. The example using H at the beginning of this section showed what could go wrong if we allowed the evaluator for any projection. As we have seen, the fact that $f : \beta^S \Rightarrow \alpha^S$ means that `f` definitely needs to evaluate the argument. If we relax this restriction, it is possible that we might violate the safety condition. For example, consider the functions defined in Figure 4, and suppose that the application

`(f 0 [0,⊥])` is being evaluated with $\xi_{T \circ H}$. The following projection information is relevant:

$$G_2 : ID \Rightarrow T \circ H$$
$$F_2 : ID \Rightarrow H$$
$$CONS_1 : T \circ H \Rightarrow ID$$
$$CONS_2 : T \circ H \Rightarrow T \circ H$$

but we do not have $G_2 : ID^S \Rightarrow (T \circ H)^S$. If we dropped the restriction, so that we had a theorem equivalent to Theorem 4.6, and the first argument to `f` was `0` and the second argument to `g` evaluated to `0`, then we could mark the expression `[0, ⊥]`, which is an argument to `g`, with $\xi_{T \circ H}$. Later, the evaluation of the application of `before` would force the evaluation of the argument with evaluator $\xi_{H_B \circ T \circ H}$ (by Theorem 4.8) $= \xi_{T \circ H}$, which will cause an infinite computation, which is not safe.

```
f x ys = if x=0 then (g E ys):(before ys)
                else []
g u vs = if u=0 then sumlist vs else 5
```

Figure 4: An example to show the necessity of neededness

This theorem says something very important about the use of ABS on shared closures. Recall that $ABS = lift\ BOT$, and ξ_{BOT} is the evaluator which fails to terminate on all expressions. Note also that BOT is a smash projection. Therefore, if we have some function f which satisfies $f : \beta^S \Rightarrow BOT^S$, that is $f : \beta^S \Rightarrow STR \circ ABS$, then the above theorem says that an infinite computation is allowed no matter what the argument to f is. This is consistent with [22], where $STR \circ ABS$ is called $FAIL$, and has the intuitive interpretation that the program will always fail to terminate. However, if we only have that $f : \beta \Rightarrow BOT$, and not $f : \beta^S \Rightarrow STR \circ ABS$, then the above theorem says that we cannot do anything with this information, and so care must be taken with the informal pragmatics given for ABS given in [22]. For example, we have given the projection information for *length* in this paper as $length : ID^S \Rightarrow T^S$. In [22], it is shown that *length* actually satisfies a stronger condition, namely that $length : ID^S \Rightarrow FIN\ ABS$. The intended interpretation of this is that if an application of `length` is evaluated, then closures do not have to be created for the elements in the list, as `length` never evaluates them. However, *drop* $(FIN\ ABS)$ is not a smash projection, and so when an argument to `length` is a shared closure, Theorem 4.8 says we can only use $\xi_{(smash\ (FIN\ ABS)) \circ \gamma}$, and $smash\ (FIN\ ABS) = T$. More generally, by the above theorem, any projection which throws away part of the structure of its argument will never have its evaluator evaluate a shared closure.

This theorem gives another motivation for developing a sharing analysis; if we know that an argument expression is not shared, then there is no need to restrict ourselves to using the evaluator corresponding to the smash part of a projection.

At the end of this section, we might rightly ask if this information is ever going to be of any use. The answer is yes, and the intuition is that the projection for an argument of a function must be one which is in some sense valid in all places where the argument appears in the body of the function. For example, the function given in Figure 5 satisfies $F_2 : ID \Rightarrow T$, but its second argument can be evaluated using $\xi_{T \circ H}$ if its first argument is non-zero. Theorem 4.8 allows us to do this.

```
f x ys = if x=0 then length ys
                else sumlist ys
```

Figure 5: An example to show the use of run-time choice of versions

5 Compilation

Theorem 4.2 says that when a function, f, is $f : \beta^S \Rightarrow \alpha^S$, then the evaluation order may be changed so that the argument to f can be evaluated with evaluator ξ_α. There are two degrees of freedom in this definition, and different choices are made in the sequential and parallel case. Firstly, there is a choice of *how the evaluation order is changed*. In the sequential implementation, we choose to evaluate the argument to f, *before* evaluating the application (sometimes called 'strictly'), whilst in a parallel implementation we choose to create a new task to evaluate the expression, so that it can be evaluated *in parallel* with the function application[7]. Secondly, there is a choice of *how to implement the evaluator* ξ_α. For example, one may choose to implement $\xi_{T \circ H}$ in a sequential implementation by evaluating the whole of the structure of the list, and each element in the list, before returning any result. This is like what happens with a Lisp implementation, and creates the least amount of heap. The compilation rules we give for the sequential machine implement the evaluators in this way, but other choices could of course have been made. A parallel implementation may choose to only evaluate the list to

[7] When an expression does not require enough evaluation steps to make it into a separate process, the argument can be evaluated before the evaluation of the application instead of in parallel with it.

WHNF, and then spawn two subtasks, one to evaluate the head to WHNF, and the other to evaluate the tail with $\xi_{T \circ H}$. This ensures that the list becomes available to a consumer process as soon as possible, and allows the greatest amount of parallelism. The compilation rules we give for the parallel machine implement the evaluators in this way, but other choices could of course have been made. Essentially, a parallel implementation chooses to gain speed by creating a closure in the heap and spawning a parallel process to evaluate it, whilst a sequential implementation gains its speed by evaluating an expression without creating a closure for it.

We will demonstrate compiling the projection transformer into code by giving compilation rules for a parallel G-machine based on the Spineless G-machine [8]. Whilst more efficient code can be generated (see [21, 2], for example), it is easy to show how to introduce the projection transformer information into code for the Spineless G-machine.

Three compilation functions will be considered:

- $\mathcal{R}$: Used to compile the bodies of function definitions. The result will be a pointer to a node in (at least) WHNF.
- $\mathcal{E}$: Used to compile code for arguments in a function application which can be evaluated before the function is applied. The result will be a pointer to a node in (at least) WHNF.
- $\mathcal{C}$: Used to generate code which creates a closure in the heap. A pointer to the closure is left on the top of the stack.

Projection transformer information is used to optimise applications of functions. Therefore we need only consider the compilation rules for functions applications, and the special case when the function being applied is `cons`. The relevant compilation rules of the Spineless G-machine are given in Figure 6.

The code generated for the R-scheme creates closures in the heap for each of the argument expressions, squeezes out the `n` arguments to the function, and pushes a pointer to the function `g`. `ENTER` checks to see if there are enough arguments on the stack to execute the code for the function pointed at by the top element of the stack; if there are, then it executes the code for the function on the top of the stack, otherwise it creates a closure and returns to the calling frame. The code produced by the $\mathcal{E}$- and $\mathcal{C}$-schemes are similar. In the $\mathcal{E}$-scheme, the `SCALL` instruction causes the evaluation of the application in a new frame on the stack; the `STORE (m+1)` instruction, used in the $\mathcal{C}$-scheme, creates a closure with `(m+1)` elements from the top of the stack, leaving a pointer to the closure on the stack. In the special case that the function being applied is `CONS`, a `CONS` node is created in the heap when the two argument expressions have been constructed.

A version of the code needs to be generated for each of the possible evaluators that have been chosen. When generating code, we need to keep in mind the discussion in Section 4.3. In order to get the full power of the evaluation transformer model in the presence of graph reduction, we need to have mechanisms for recording the amount of evaluation that has been requested so far, for updating this with any further requests, and for choosing the appropriate version of a function when the evaluation begins. This is discussed in Section 5.3. In Sections 5.1 and 5.2 we take a simpler approach, where we only use the information when a function is applied to another function application in the source text of the program, which means we can choose the appropriate version of the function entirely at compile-time. Section 5.1 shows how to generate code for a sequential machine, and Section 5.2 for a parallel machine. Note that the code generated in Section 5.3 is sometimes called 'run-time choice of versions' and the other 'compile-time choice of versions'.

In the subsequent subsections, we will signify that code is being generated for evaluator ξ_β by subscripting the compilation rule with β.

5.1 Compilation for Sequential Machines

In a sequential implementation, we use the projection information to avoid building closures.

In the following compilation rules, we assume that the function `g` has k arguments, and that the projection transformers for $lift\ g$ are G_1 to G_k. Where `g` is applied to m arguments, and $m > k$, then for all $i > k$, G_i returns $lift\ ID$ when applied to any projection.

By $\mathtt{g}_\beta$ we will denote the code entry point for the version of `g` which is used when evaluating an application of `g` with evaluator ξ_β. `PUSHGLOBAL` $\mathtt{g}_\beta$ pushes a pointer to this code address onto the stack. This is the compile-time choice of version for the function.

The compilation rules in Figure 7 can be justified by reference to Theorems 4.2 and 4.6. When $lift\ g : \beta^S \Rightarrow \alpha^S$ for its ith argument, Theorem 4.2 says that the evaluation order can be changed so that the ith argument is evaluated with evaluator ξ_α. This is captured by compiling the expression with $\mathcal{E}_\alpha$. If this is not the case, but $g : \beta \Rightarrow \alpha$, Theorem 4.6 says that a closure can be created in the heap such that when the expression is evaluated, it will be evaluated with ξ_α. This is captured by compiling the expression with the $\mathcal{C}_\alpha$-compilation rule.

Note that, as Theorem 4.6 allows, the evaluation information is propagated through the closures which are stored in the heap.

$$
\begin{array}{lll}
\mathcal{R}\ [\![\ \texttt{g D1 ... Dm}\ ]\!]\ \texttt{r n} = & \mathcal{C}\ [\![\ \texttt{Dm}\ ]\!]\ \texttt{r n};\ \ldots;\ \mathcal{C}\ [\![\ \texttt{D1}\ ]\!]\ \texttt{r (n+m}\perp\texttt{1)}; \\
& \texttt{SQUEEZE m n; PUSHGLOBAL g; ENTER} \\
\mathcal{E}\ [\![\ \texttt{g D1 ... Dm}\ ]\!]\ \texttt{r n} = & \mathcal{C}\ [\![\ \texttt{Dm}\ ]\!]\ \texttt{r n};\ \ldots;\ \mathcal{C}\ [\![\ \texttt{D1}\ ]\!]\ \texttt{r (n+m}\perp\texttt{1)}; \\
& \texttt{PUSHGLOBAL g; SCALL (m+1)} \\
\mathcal{C}\ [\![\ \texttt{g D1 ... Dm}\ ]\!]\ \texttt{r n} = & \mathcal{C}\ [\![\ \texttt{Dm}\ ]\!]\ \texttt{r n};\ \ldots;\ \mathcal{C}\ [\![\ \texttt{D1}\ ]\!]\ \texttt{r (n+m}\perp\texttt{1)}; \\
& \texttt{PUSHGLOBAL g; STORE (m+1)}
\end{array}
$$

Rules for General Function Applications

$$
\begin{array}{lll}
\mathcal{R}\ [\![\ \texttt{cons D1 D2}\ ]\!]\ \texttt{r n} = & \mathcal{C}\ [\![\ \texttt{D2}\ ]\!]\ \texttt{r n};\ \mathcal{C}\ [\![\ \texttt{D1}\ ]\!]\ \texttt{r (n+1)}; \\
& \texttt{CONS; SQUEEZE 1 n; RETURN} \\
\mathcal{E}\ [\![\ \texttt{cons D1 D2}\ ]\!]\ \texttt{r n} = & \mathcal{C}\ [\![\ \texttt{D2}\ ]\!]\ \texttt{r n};\ \mathcal{C}\ [\![\ \texttt{D1}\ ]\!]\ \texttt{r (n+1)}; \\
& \texttt{CONS} \\
\mathcal{C}\ [\![\ \texttt{cons D1 D2}\ ]\!]\ \texttt{r n} = & \mathcal{C}\ [\![\ \texttt{D2}\ ]\!]\ \texttt{r n};\ \mathcal{C}\ [\![\ \texttt{D1}\ ]\!]\ \texttt{r (n+1)}; \\
& \texttt{CONS}
\end{array}
$$

Rules for `cons`

Figure 6: Compilation Rules for the Spineless G-machine

$$
\mathcal{A}_i = \begin{cases} \mathcal{E}_\alpha & \text{if } \ G_i(\beta^S) = \alpha^S \\ \mathcal{C}_\alpha & \text{if } \ G_i(\beta^S) = lift\ \alpha \text{ and } G_i(\beta^S) \neq \alpha^S \end{cases}
$$

$$
\gamma_i = drop\ (G_i\ (lift\ \beta^S))
$$

$$
\begin{array}{lll}
\mathcal{R}_\beta\ [\![\ \texttt{g D1 ... Dm}\ ]\!]\ \texttt{r n} = & \mathcal{A}_m\ [\![\ \texttt{Dm}\ ]\!]\ \texttt{r n};\ \ldots;\ \mathcal{A}_1\ [\![\ \texttt{D1}\ ]\!]\ \texttt{r (n+m}\perp\texttt{1)}; \\
& \texttt{SQUEEZE m n; PUSHGLOBAL g}_\beta\texttt{; ENTER} \\
\mathcal{E}_\beta\ [\![\ \texttt{g D1 ... Dm}\ ]\!]\ \texttt{r n} = & \mathcal{A}_m\ [\![\ \texttt{Dm}\ ]\!]\ \texttt{r n};\ \ldots;\ \mathcal{A}_1\ [\![\ \texttt{D1}\ ]\!]\ \texttt{r (n+m}\perp\texttt{1)}; \\
& \texttt{PUSHGLOBAL g}_\beta\texttt{; SCALL (m+1)} \\
\mathcal{C}_\beta\ [\![\ \texttt{g D1 ... Dm}\ ]\!]\ \texttt{r n} = & \mathcal{C}_{\gamma_m}\ [\![\ \texttt{Dm}\ ]\!]\ \texttt{r n};\ \ldots;\mathcal{C}_{\gamma_1}\ [\![\ \texttt{D1}\ ]\!]\ \texttt{r (n+m}\perp\texttt{1)}; \\
& \texttt{PUSHGLOBAL g}_\beta\texttt{; STORE (m+1)};
\end{array}
$$

Rules for General Function Applications

$$
\begin{array}{lll}
\mathcal{R}_{ID}\ [\![\ \texttt{cons D1 D2}\ ]\!]\ \texttt{r n} = & \mathcal{C}_{ID}\ [\![\ \texttt{D2}\ ]\!]\ \texttt{r n};\ \mathcal{C}_{ID}\ [\![\ \texttt{D1}\ ]\!]\ \texttt{r (n+1)}; \\
& \texttt{CONS;SQUEEZE 1 n; RETURN} \\
\mathcal{R}_T\ [\![\ \texttt{cons D1 D2}\ ]\!]\ \texttt{r n} = & \mathcal{E}_T\ [\![\ \texttt{D2}\ ]\!]\ \texttt{r n};\ \mathcal{C}_{ID}\ [\![\ \texttt{D1}\ ]\!]\ \texttt{r (n+1)}; \\
& \texttt{CONS;SQUEEZE 1 n; RETURN} \\
\mathcal{R}_H\ [\![\ \texttt{cons D1 D2}\ ]\!]\ \texttt{r n} = & \mathcal{C}_H\ [\![\ \texttt{D2}\ ]\!]\ \texttt{r n};\ \mathcal{E}_{ID}\ [\![\ \texttt{D1}\ ]\!]\ \texttt{r (n+1)}; \\
& \texttt{CONS;SQUEEZE 1 n; RETURN} \\
\mathcal{R}_{H_B}\ [\![\ \texttt{cons D1 D2}\ ]\!]\ \texttt{r n} = & \mathcal{C}_{ID}\ [\![\ \texttt{D2}\ ]\!]\ \texttt{r n};\ \mathcal{E}_{ID}\ [\![\ \texttt{D1}\ ]\!]\ \texttt{r (n+1)}; \\
& \texttt{CONS;SQUEEZE 1 n; RETURN} \\
\mathcal{R}_{T\circ H}\ [\![\ \texttt{cons D1 D2}\ ]\!]\ \texttt{r n} = & \mathcal{E}_{T\circ H}\ [\![\ \texttt{D2}\ ]\!]\ \texttt{r n};\ \mathcal{E}_{ID}\ [\![\ \texttt{D1}\ ]\!]\ \texttt{r (n+1)}; \\
& \texttt{CONS;SQUEEZE 1 n; RETURN}
\end{array}
$$

$\mathcal{R}$-rules for `cons`

Figure 7: Compilation Rules for a Sequential Machine

In our examples, the only structured data type that we consider is the domain of lists, and our example projections are ID, T, H, H_B and $T \circ H$. The compilation rules given in Figure 7 for `cons` implement the sequential pragmatics we have associated with the evaluators which correspond with these projections. For example, the compilation rule $\mathcal{R}_{T \circ H}$ evaluates the tail of the list using $\xi_{T \circ H}$, then evaluates the head using ξ_{ID}, and then creates a `CONS` cell pointing to the evaluated head and tail. Note the difference between th compilation rules $\mathcal{R}_H$ and $\mathcal{R}_{H_B}$. When an application of `cons` is being evaluated with ξ_H, a closure is created for the tail which will cause it to be evaluated with ξ_H, if it is ever evaluated, whilst evaluation with ξ_H creates a closure for the tail which will only be evaluated with ξ_{ID}, if it is ever evaluated. The $\mathcal{E}$-scheme for applications of `cons` are the same as the $\mathcal{R}$-scheme, except there are no `SQUEEZE` or `RETURN` instructions. In a similar manner, the $\mathcal{C}$-scheme for applications of `cons` are obtained from the $\mathcal{R}$-scheme by removing the `SQUEEZE` and `RETURN` instructions, and replacing uses of $\mathcal{E}_\alpha$ with uses of $\mathcal{C}_\alpha$.

5.2 Compilation for Parallel Machines

When a change in the evaluation order is allowed, the parallel code generates a new task to evaluate the argument expression. Evaluators are implemented by evaluating nodes to WHNF, and then possibly spawning new tasks to evaluate the subexpressions of the (`CONS`) node that has been created.

For the parallel machine, the $\mathcal{C}_\beta$-scheme is exactly the same as for the sequential machine. The form of the $\mathcal{R}$-scheme is exactly the same as the sequential case, except that it uses the $\mathcal{P}$(arallel)-scheme, given in Figure 8, for compiling subexpressions which would have been compiled with the $\mathcal{E}$-scheme in the compilation rules for the sequential machine. The `SPAWN` instruction creates a process to evaluate the closure pointed to by the top element on the stack. The $\mathcal{P}$-scheme rules for `cons` have the same structure as the $\mathcal{E}$-scheme, where all uses of $\mathcal{E}_\alpha$ are replaced by $\mathcal{P}_\alpha$. An example is given in Figure 8. Note that this defines the evaluator $\xi_{T \circ H}$ for the parallel implementation so that it creates closures for each of the tail and head of the list, and spawns processes to evaluate them with appropriate evaluators.

5.3 Run-Time Choice of Versions

The function definition in Figure 5 is an example of where information available at run-time can allow us to choose a stronger evaluator for an expression than the one which could be chosen just using compile-time information. In the example, if the first argument to `f` is non-zero, then the second argument can be evaluated with $\xi_{T \circ H}$ rather than just ξ_T because it is an argument to `sumlist`. In Section 4.3 we saw that this situation arises when functions in the body of a function definition are applied to formal parameters of the function being compiled. Theorem 4.8 said that we can update the evaluator on the argument expression as long as we can guarantee that the expression is going to be evaluated, and that the new projection is a smash projection. This corresponds to giving new compilation rules for compiling variables for the $\mathcal{R}_\beta$, $\mathcal{E}_\beta$ and $\mathcal{P}_\beta$ compilation schemes; the other rules are unchanged. Because the theorem only allows the evaluator to be updated when the expression definitely needs to be evaluated, and because the $\mathcal{C}_\beta$ rule is used when we do not know if an expression has to be evaluated, it will be exactly the same as the Spineless G-machine rule. The new compilation rules are given in Figure 9.

The compilation rules that are given in Figure 9 introduce some new instructions. We will only outline their meaning in this paper; similar instructions have been fully defined elsewhere, [5, 15] for example, albeit for a simpler set of evaluators.

In order to support the run-time updating of versions, the implementation must have some way of accessing and updating the evaluator stored on an expression. Suppose that a node is marked with the evaluator ξ_γ. The instruction `UPDATEEV` ξ_α causes the the evaluator on the node to be updated with $\xi_{\alpha \circ \gamma}$. Both `SCALL` and `EVAL` cause the evaluation of an expression. Now they must inspect the evaluator on the node before evaluation begins, and choose the correct version of the function to be executed. The `SPAWN` instruction should only create a task the first time a closure is `SPAWN`ed.

There is one more point that we have glossed over. In the above we said that `UPDATEEV` ξ_α caused the updating of the evaluator with $\xi_{\alpha \circ \gamma}$. The problem is that our implementation may not support this evaluator. Here we evaluate the expression first with the version corresponding to ξ_γ, and then with the version corresponding to ξ_a. By Lemma 2.8, this does more evaluation than using the evaluators in the opposite order. Note also that it is better to use $\xi_{\alpha \circ \gamma}$ if it exists, because it may save the building of some closures, or allow more opportunities for parallelism. For example, following ξ_H with ξ_T in a sequential implementation means that ξ_H will always create a closure for the tail, whereas $\xi_{T \circ H}$ does not need to do so. To support this, we have to keep a number of different evaluators on a node, and the number probably has some correspondence with the width of the lattice of evaluators supported by the implementation.

$$\mathcal{A}_i = \begin{cases} \mathcal{P}_\alpha & \text{if} \quad G_i(\beta^S) = \alpha^S \\ \mathcal{C}_\alpha & \text{if} \quad G_i(\beta^S) = lift\ \alpha \text{ and } G_i(\beta^S) \neq \alpha^S \end{cases}$$

$$\begin{aligned} \mathcal{P}_\beta\ [\![\ \texttt{g D1 ... Dm}\ ]\!]\ \texttt{r n} \quad = \quad & \mathcal{A}_m\ [\![\ \texttt{Dm}\ ]\!]\ \texttt{r n};\ \ldots; \mathcal{A}_1\ [\![\ \texttt{D1}\ ]\!]\ \texttt{r}\ (\texttt{n} + \texttt{m} \perp \texttt{1}); \\ & \texttt{PUSHGLOBAL g}_\beta;\ \texttt{STORE}\ (\texttt{m} + \texttt{1});\ \texttt{SPAWN} \end{aligned}$$

Rule for General Function Applications

$$\begin{aligned} \mathcal{R}_{T \circ H}\ [\![\ \texttt{cons D1 D2}\ ]\!]\ \texttt{r n} \quad = \quad & \mathcal{P}_{T \circ H}\ [\![\ \texttt{D2}\ ]\!]\ \texttt{r n};\ \mathcal{P}_{ID}\ [\![\ \texttt{D1}\ ]\!]\ \texttt{r}\ (\texttt{n} + \texttt{1}); \\ & \texttt{CONS;SQUEEZE 1 n;}\ \texttt{RETURN} \end{aligned}$$

An example $\mathcal{R}$-rule for `cons`

Figure 8: Example Compilation Rules for a Parallel Machine

$$\begin{aligned} \mathcal{R}_\beta\ [\![\ \texttt{x}\ ]\!]\ \texttt{r n} \quad &= \quad \texttt{PUSH}\ (\texttt{n} \perp (\texttt{r x}));\ \texttt{UPDATEEV}\ (smash\ \xi_\beta);\texttt{EVAL} \\ \mathcal{E}_\beta\ [\![\ \texttt{x}\ ]\!]\ \texttt{r n} \quad &= \quad \texttt{PUSH}\ (\texttt{n} \perp (\texttt{r x}));\ \texttt{UPDATEEV}\ (smash\ \xi_\beta);\texttt{SCALL 1} \\ \mathcal{P}_\beta\ [\![\ \texttt{x}\ ]\!]\ \texttt{r n} \quad &= \quad \texttt{PUSH}\ (\texttt{n} \perp (\texttt{r x}));\ \texttt{UPDATEEV}\ (smash\ \xi_\beta);\texttt{SPAWN} \\ \mathcal{C}_\beta\ [\![\ \texttt{x}\ ]\!]\ \texttt{r n} \quad &= \quad \texttt{PUSH}\ (\texttt{n} \perp (\texttt{r x})) \end{aligned}$$

Figure 9: Compilation Rules for Variables for Run-time Choice of Versions

$$\mathcal{R}_\beta\ [\![\ \texttt{if D1 D2 D3}\ ]\!]\ \texttt{r n} = \mathcal{E}_{ID}[\![\ \texttt{D1}\ ]\!]\ \texttt{r n;JFALSE L1;}\mathcal{R}_\beta\ [\![\ \texttt{D2}\ ]\!]\ \texttt{r n;LABEL L1;}\mathcal{R}_\beta\ [\![\ \texttt{D3}\ ]\!]\ \texttt{r n}$$

Figure 10: $\mathcal{R}_\beta$ rule for compiling the conditional

5.4 Compilation of Conditional and Case Expressions

Conditional and case expressions have the property that after selecting the appropriate expression to be evaluated, the evaluator which evaluates the selected expression is the evaluator which is evaluating the whole expression, even though the evaluator for the selecting expression may have been different. We proved this by transformational reasoning in [5]. As an example, we give the $\mathcal{R}_\beta$ rule for compiling a conditional in Figure 10.

6 Tightening Up the Correctness Proofs

The proofs we have given for Theorems 4.2, 4.6 and 4.8 required some hand-waving. When we have non-smash projections, we cannot just say that evaluators are reduction strategies, and then rely on the fact that reduction preserves meaning, the fact that the Church-Rosser property ensures terminating reduction sequences give the same answer, and the fact that we preserve termination, in order to prove that our reduction model is correct. As we saw in Definition 3.3, the evaluators corresponding to non-smash projections may also change the meaning of expression they are evaluating, even when they terminate. Moreover, we have written of techniques for dealing with graph reduction, without formally proving our theorems against a model of graph reduction. Even worse, we did not even try to prove that the compilation rules given in Section 5 correctly implemented the evaluation model! However, we believe that all these things are possible, and have begun work on proving them. The major problem seems to be to devise some operational semantics so that we can reason about changing the value of an expression using an evaluator. Proofs about graph reduction should hopefully then follow through as in [1], and we should be able to appeal to [14] to get the basis for the proof of the correctness of our compilation rules.

7 Implementing the Model

We defined a parallel abstract machine which supports this model in the special case that only the evaluators ξ_{ID}, ξ_T and $\xi_{T \circ H}$ are supported, and the versions of functions are chosen at run-time. Full compilation rules and the definition of a first version of the machine are contained in [5]; a later version of the machine is given as a Miranda[8] program in [15]. An implementation of this abstract machine on a simple network of transputers, with the abstract machine code being macro-expanded to transputer machine code, has been completed [13].

Loogen has also defined a parallel abstract machine based on the same model, which has been implemented in occam on a network of transputers [16].

[8] Miranda is a trademark of Research Software Ltd.

8 Conclusion

The evaluation transformer reduction model is a natural extension of lazy evaluation when we have information about how arguments of functions are going to be used. This information is available from projection analysis [22] and abstract interpretation [3, 4], and in this paper, we have shown how to obtain it from projection analysis. We have shown how to use this information in compiling code for sequential and parallel implementations, and have shown clearly the distinction between run-time and compile-time choice of versions for functions; the former arises from using extra information about the application of functions in the body of a function to its formal parameters.

9 Acknowledgements

I am grateful to Phil Wadler and John Hughes for their provocative remarks in [22] concerning the relative power of abstract interpretation and projection analysis, which initiated this work. Since then, I have had a number of useful discussions with Phil Wadler concerning the relationship between projection analysis and abstract interpretation. Whilst I have been working at Imperial College, I have appreciated long discussions with Sebastian Hunt and David Sands about this work, especially about the definition of evaluators. They are still not happy with the definitions (and neither am I)! I thank them for their helpful comments on various drafts of this paper.

This work was partially completed whilst the author was working for the GEC Hirst Research Centre, East Lane, Wembley, Middlesex HA9 7PP, UK, and so was partially funded by ESPRIT Project 415: Parallel Architectures and Languages for AIP – A VLSI-Directed Approach. In my time working at GEC I greatly appreciated the discussions I had with David Lester, and I also thank him for his comments on this paper.

References

[1] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Term graph rewriting. In *Proceedings*

of PARLE 87, volume 2, pages 141–158. Springer-Verlag LNCS 259, Eindhoven, The Netherlands, June 1987.

[2] A. Bloss, P. Hudak, and J. Young. Code optimisations for lazy evaluation. In *Proceedings of the Workshop on the Implementation of Lazy Functional Languages*, Aspenäs, Göteborg, Sweden, 5–8 September 1988.

[3] G. L. Burn. Evaluation transformers – A model for the parallel evaluation of functional languages (extended abstract). In G. Kahn, editor, *Proceedings of the Functional Programming Languages and Computer Architecture Conference*, pages 446–470. Springer-Verlag LNCS 274, September 1987.

[4] G.L. Burn. *Abstract Interpretation and the Parallel Evaluation of Functional Languages*. PhD thesis, Imperial College, University of London, March 1987.

[5] G.L. Burn. A shared memory parallel G-machine based on the evaluation transformer model of computation. In *Proceedings of the Workshop on the Implementation of Lazy Functional Languages*, pages 301–330, Aspenäs, Göteborg, Sweden, 5–8 September 1988.

[6] G.L. Burn. A relationship between abstract interpretation and projection analysis. In *17th Annual ACM Symposium on the Principles of Programming Languages*, pages 151–156, San Francisco, 17–19 January 1990. ACM.

[7] G.L. Burn, C.L. Hankin, and S. Abramsky. Strictness analysis for higher-order functions. *Science of Computer Programming*, 7:249–278, November 1986.

[8] G.L. Burn, S.L. Peyton Jones, and J.D. Robson. The spineless G-machine. In *Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, pages 244–258, Snowbird, Utah, 25–27 July 1988.

[9] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Sixth Annual Symposium on Principles of Programming Languages*, pages 269–282. ACM, January 1979.

[10] P. Dybjer. Inverse image analysis. In *Proceedings of the 14th International Colloquium on Automata, Languages and Programming*, Karlsruhe, Germany, July 1987. Springer-Verlag LNCS 267.

[11] C.V. Hall and D.S. Wise. Compiling strictness into streams. In *14th Annual ACM Symposium on the Principles of Programming Languages*, pages 132–143. ACM, January 1987.

[12] P. Hudak and J. Young. Higher order strictness analysis in untyped lambda calculus. In *Proceedings of 12th ACM Symposium on Principles of Programming Languages*, pages 97–109, January 1986.

[13] H. Kingdon, D.R. Lester, and G.L. Burn. A transputer-based HDG–machine. *The Computer Journal*, 34(4):290–301, August 1991.

[14] D.R. Lester. *Combinator Graph Reduction: A Congruence and its Applications*. DPhil thesis, Oxford University, 1988. *Also* published as Technical Monograph PRG-73.

[15] D.R. Lester and G.L. Burn. An executable specification of the HDG-Machine. In *Workshop on Massive Parallelism: Hardware, Programming and Applications*, Amalfi, Italy, 9–15 October 1989.

[16] R. Loogen, H. Kuchen, K. Indermark, and W. Damm. Distributed implementation of programmed graph reduction. In E. Odijk, M. Rem, and J.-C Syre, editors, *Proceedings of PARLE 89*, volume 1, pages 136–157, Eindhoven, The Netherlands, 12–16 June 1989. SPRINGER-Verlag LNCS 365.

[17] D Maurer. Strictness computation using special lambda-expressions. In H Ganzinger and N.D. Jones, editors, *Proceedings of the Workshop on Programs as Data Objects*, pages 136–155, DIKU, Copenhagen, Denmark, 17-19 October 1985. Springer-Verlag LNCS 217.

[18] A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. PhD thesis, University of Edinburgh, Department of Computer Science, December 1981. *Also* published as CST-15-81.

[19] F. Nielson. *Abstract Interpretation Using Domain Theory*. PhD thesis, Department of Computer Science, University of Edinburgh, October 1984. *Also* CST-31-84.

[20] F. Nielson. Two-level semantics and abstract interpretation. *Theoretical Computer Science*, 69:117–242, 1989.

[21] S.L. Peyton Jones and J. Salkild. The Spineless Tagless G-Machine. In D. B. MacQueen, editor, *Proceedings of the Functional Programming Languages and Computer Architecture Conference*, pages 184–201. ACM, 11–13 September 1989.

[22] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In G. Kahn, editor, *Proceedings of the Functional Programming Languages and Computer Architecture Conference*, pages 385–407. Springer-Verlag LNCS 274, September 1987.