Compiling Laziness using Projections

Ross Paterson

School of Computing, University of North London, Eden Grove, London N7 8DB, UK

Abstract. Projection-based strictness analysis is a powerful technique, able to cope with non-flat domains and latent demand. By analysing the projections as embedding-projection pairs, we develop an algorithm to translate lazy functions into a strict functional language with explicit closures. The translated functions typically operate on simpler types than the originals, in particular types containing fewer liftings, which correspond to the operational notion of closures. Like the analysis on which it is based, our algorithm is restricted to first-order functions.

1 Introduction

A lazy functional language provides great expressive power, but this comes at a cost in both space and time, for delayed computations must be stored as closures on the heap. The management of these closures also adds to execution time, and the frequent context switches frustrate both compile-time optimization and efficient exploitation of hardware.

Strictness analysis [17,3,20,14] aims to ameliorate this situation by identifying strict functions, which can be implemented using call-by-value. The arguments of such functions may be evaluated early and passed unboxed. This has contributed to compiler performance, but the changes to the code generator are seldom formally verified.

There have been some efforts to verify these alterations. Modifications of the CPS translation [2] to use simple strictness information have been described by Danvy and Hatcliffe [6] and by Burn and Le Métayer [4]. Amtoft [1] describes a modification of a type-based strictness analysis to generate code that creates fewer thunks.

Peyton Jones and Launchbury [19,18] introduce flat domains to their source language, interpreting them operationally as unboxed values. They are thus able to verify unboxing transformations in the source language.

The above systems are based on information about active demand, i.e. statements that a certain amount of a data structure will be required. In the presence of lazy data structures, e.g. streams, one is also interested in latent demand, i.e. statements that though a value may not be required, if it is then so much of it must be evaluated. This sort of strictness information can be obtained by projection-based strictness analysis [20].

This paper extends the projection technique to obtain a new algorithm translating lazy functional programs to more efficient strict forms.

1.1 Projection-based Strictness Analysis

In order to capture strictness over simple domains as well as non-flat domains, Wadler and Hughes [20] introduced projections over extended domains to describe evaluation contexts. An equivalent formulation [15] defines a projection over a type t as a partial function $\alpha : t \circ\!\!\to t$ satisfying $\alpha \leq id$ and $\alpha \circ \alpha = \alpha$. For example, the projection $\mathsf{Str} : \forall\, a.\, a_\perp \circ\!\!\to a_\perp$ is defined only if its argument is not $\perp$. (A type $t_\perp$ consists of the elements of t, which may or may not contain a bottom element, plus a new bottom element.) The projection $\mathsf{Both} : \forall\, a, b.\, a_\perp \times b_\perp \to a_\perp \times b_\perp$ is defined only if neither of its arguments are $\perp$.

The analysis of a total function $f : t_\alpha \to t_\beta$ for a given context projection $\beta : t_\beta \circ\!\!\to t_\beta$ yields an equation

$$\beta \circ f = \beta \circ f \circ \alpha \tag{1}$$

This equation implies that for an application of f in the context β, the context of the argument of f is given by the projection $\alpha : t_\alpha \circ\!\!\to t_\alpha$. For example, the iterative factorial function $\mathit{fact} : \mathsf{Num}_\perp \times \mathsf{Num}_\perp \to \mathsf{Num}_\perp$ (where Num is the set of numbers, without $\perp$) is seen to satisfy

$$\mathsf{Str} \circ \mathit{fact} = \mathsf{Str} \circ \mathit{fact} \circ \mathsf{Both} \tag{2}$$

indicating that it is strict in both arguments. Moreover, such projection equations can also capture latent strictness properties such as *head strictness*, which says that if a list is ever evaluated to weak head normal form, its head will also be evaluated. This analysis has been implemented by Kubiak, Hughes and Launchbury [13]. The present formulation is based on theirs.

1.2 This Paper

The point of departure of this paper is Launchbury's observation [15] that the images of these partial projections correspond to unboxed types. These images may be described using embedding-projection pairs. A partial projection α is factored as $\alpha^{\mathrm{E}} \circ \alpha^{\mathrm{P}}$, for a total function $\alpha^{\mathrm{E}} : t'_\alpha \to t_\alpha$ and a partial function $\alpha^{\mathrm{P}} : t_\alpha \circ\!\!\to t'_\alpha$ such that $\alpha^{\mathrm{P}} \circ \alpha^{\mathrm{E}} = id$ and $\alpha^{\mathrm{E}} \circ \alpha^{\mathrm{P}} \leq id$. The type t'_α will be a simpler variant of the type t_α. For example, the projection Str factors as an embedding of a type a in the type $a_\perp$. The latter lifted type might be implemented using closures, the former using unboxed values. Here the embedding component merely boxes the number; the projection component demands the value of a closure, and is therefore partial. Similarly, the projection Both corresponds to an embedding of the type $a \times b$ in the type $a_\perp \times b_\perp$.

Now (2) expands to

$$\mathsf{Str}^{\mathrm{E}} \circ \mathsf{Str}^{\mathrm{P}} \circ \mathit{fact} = \mathsf{Str}^{\mathrm{E}} \circ \mathsf{Str}^{\mathrm{P}} \circ \mathit{fact} \circ \mathsf{Both}^{\mathrm{E}} \circ \mathsf{Both}^{\mathrm{P}}$$

Composing $\mathsf{Str}^{\mathrm{P}}$ on the left to cancel the $\mathsf{Str}^{\mathrm{E}}$'s, this is equivalent to

$$\mathsf{Str}^{\mathrm{P}} \circ \mathit{fact} = \mathsf{Str}^{\mathrm{P}} \circ \mathit{fact} \circ \mathsf{Both}^{\mathrm{E}} \circ \mathsf{Both}^{\mathrm{P}} \tag{3}$$

Now we can fuse the composition $\mathsf{Str}^{\mathrm{P}} \circ \mathit{fact} \circ \mathsf{Both}^{\mathrm{E}}$, yielding a variant fact' : $\mathsf{Num} \times \mathsf{Num} \circ\!\!\rightarrow \mathsf{Num}$. This function, which operates on unboxed numbers instead of closures, can then be used in place of *fact*. The projection component $\mathsf{Both}^{\mathrm{P}}$ goes on to provide a context for the arguments of *fact*.

General fusion is difficult, so we shall alter projection analysis to generate equations of the form

$$\beta^{\mathrm{P}} \circ f = f_\beta \circ \alpha^{\mathrm{P}} \tag{4}$$

for some partial function $f_\beta : t'_\alpha \circ\!\!\rightarrow t'_\beta$. The rules of our algorithm thus correspond to those of projection-based strictness analysis; the novelty here is the translated version f_β.

The source language of our translation is a small first-order lazy functional language, defined in Sect. 2. The main unusual feature of this language is that lifting is treated as a separate type constructor. This allows us to treat the issues raised by each type constructor separately, and also to deal with unlifted types, which are avoided in most accounts of strictness analysis.

The translation sketched above produces partial functions. The projection components of e-p pairs are also partial functions. To express these, we introduce in Sect. 3 a small first-order strict functional language, with explicit **delay** and *force* constructs, as in Scheme [5].

The usual next step would be to give denotational descriptions of these two languages in a semantic meta-language. However, this meta-language would be essentially equivalent to our source language. Instead, we give an axiomatic description of the lazy language, and define the strict language by a translation to the lazy language. Then the rules of our algorithm expand to formulas over the lazy language.

Since everything is translated back to the lazy language for proof purposes, we could have described our algorithm as a source-to-source transformation on the lazy language. Indeed this was done in an early version of this work, but the transformation rules were far more complex than those presented here. Also, in the strict language closures are exactly the elements of lifted types; in the lazy language the correspondence is not as precise. The strict language is also well suited to the description of sequential computations; failure of any evaluation causes failure of the whole machine. This language may also be used as an interface to existing compilers.

A translation to a strict language driven by strictness analysis is also used by Faxén [9], but there only simple strictness is used. Thunk elimination and unboxing are performed at a later stage using a type-based flow inference [8].

In Sect. 4, we show how the contexts of conventional projection analysis may be factored as embedding-projection pairs. The embedding components, being total functions, are described by the lazy language, while the projection components, being partial, are described in the strict language.

The rules of our translation algorithm are given in Sect. 5, with some examples. Section 6 concludes.

$$
\begin{array}{ll}
\text{Types} & t = \varnothing \;\mid\; \mathbb{1} \;\mid\; t \times t \;\mid\; t + t \;\mid\; t_\perp \;\mid\; a \;\mid\; \mu\, a.\, t \\[1ex]
\text{Definitions} & \mathit{def} = f_d \triangleq \lambda\, x.\, e \\[1ex]
\text{Expressions} & e = x \;\mid\; \perp \;\mid\; () \;\mid\; (e_1, e_2) \;\mid\; f\ e \\
& \quad \mid\ \textbf{case}\ e\ \textbf{of}\ \mathit{inl}\ x_1 \Rightarrow e_1 \parallel \mathit{inr}\ x_2 \Rightarrow e_2 \\
& \quad \mid\ \textbf{case}\ e\ \textbf{of}\ \mathit{lift}\ x \Rightarrow e' \\[1ex]
\text{Functions} & f = f_d \\
& \quad \mid\ \lambda\, x.\, e \\
& \quad \mid\ \mathit{fst} \;\mid\; \mathit{snd} \;\mid\; \mathit{inl} \;\mid\; \mathit{inr} \;\mid\; \mathit{lift} \;\mid\; \mathit{id} \;\mid\; \mathit{empty}
\end{array}
$$

Fig. 1. First-order lazy functional language

$$
\begin{array}{l}
\mathit{products} : \mathsf{Num}_\perp \times \mathsf{List\ Num}_\perp \rightarrow \mathsf{List\ Num}_\perp \\
\mathit{products}\ (x, \mathit{ys}) \triangleq \textbf{case}\ \mathit{ys}\ \textbf{of}\ \mathit{lift}\ n \Rightarrow \\
\qquad \textbf{case}\ n\ \textbf{of}\ \mathit{inl}\ () \Rightarrow \mathit{lift}\ (\mathit{inl}\ ()) \parallel \\
\qquad\qquad \mathit{inr}\ (y, \mathit{ys}') \Rightarrow \textbf{let}\ x' \triangleq x * y\ \textbf{in} \\
\qquad\qquad\qquad \mathit{lift}\ (\mathit{inr}\ (x', \mathit{products}\ (x', \mathit{ys}')))
\end{array}
$$

$$
\begin{array}{l}
\mathit{append} : \forall\, a.\, \mathsf{List}\ a \times \mathsf{List}\ a \rightarrow \mathsf{List}\ a \\
\mathit{append}\ (\mathit{xs}, \mathit{ys}) \triangleq \textbf{case}\ \mathit{xs}\ \textbf{of}\ \mathit{lift}\ n \Rightarrow \\
\qquad \textbf{case}\ n\ \textbf{of}\ \mathit{inl}\ () \Rightarrow \mathit{ys} \parallel \\
\qquad\qquad \mathit{inr}\ (x, \mathit{xs}') \Rightarrow \mathit{lift}\ (\mathit{inr}\ (x, \mathit{append}\ (\mathit{xs}', \mathit{ys})))
\end{array}
$$

Fig. 2. Example lazy functions

2 A First-order Lazy Language

The source language of our translation algorithm is a simple first-order lazy functional language, whose grammar is given in Fig. 1. This language is the first-order fragment of the language of [16]. The most unusual feature is that not all types have a least element $\perp$. For example, we shall see below that sums are disjoint unions, and we shall later introduce flat types of numbers and so on.

A type is said to be *pointed* if it has a least element $\perp$. The type $\mathbb{1}$ is pointed; $t_1 \times t_2$ is pointed if both t_1 and t_2 are; $t_\perp$ is always pointed; $\mu\, a.\, t$ is pointed if t is pointed for any a. Note that $t_1 + t_2$ is never pointed; nor is $\varnothing$. The result types of recursive functions, of **case** e **of** *lift* $x \Rightarrow e'$, and of $\perp$ itself, are required be pointed.

Recursive types are considered to be equivalent to their unfoldings. Recursive forms whose initial solution is empty, like $\mu\, c.\, a \times c$, are forbidden. For example, the list type constructor is defined by

$$\mathsf{List}\ a \triangleq \mu\, c.\, (\mathbb{1} + a \times c)_\perp$$

Figure 2 contains some example of functions written in this language. (We have used a little syntactic sugar to render the pattern matching more readable.) The function *products* is a typical stream processor, mapping a list of numbers to a list of partial products.

Instead of giving a denotational semantics for the language, we shall give a series of axioms that it satisfies. Firstly, the empty type $\varnothing$ has no elements:

$$\forall\, x : \varnothing.\, \mathsf{false}$$

There is however a function $\mathit{empty} : \forall\, a.\, \varnothing \to a$.

Next, the unit type $\mathbb{1}$ has a single element $()$:

$$\forall\, x : \mathbb{1}.\, x = ()$$

The product type constructor describes a true product:

$$\mathit{fst}\ (x, y) = x \qquad \mathit{snd}\ (x, y) = y \qquad \forall\, p : a \times b.\, p = (\mathit{fst}\ p, \mathit{snd}\ p)$$

The sum type constructor makes disjoint unions:

$$\begin{array}{l}(\mathbf{case}\ \mathit{inl}\ x\ \mathbf{of}\ \mathit{inl}\ x' \Rightarrow e_1 \parallel \mathit{inr}\ y' \Rightarrow e_2) = e_1[x' \mapsto x] \\ (\mathbf{case}\ \mathit{inr}\ y\ \mathbf{of}\ \mathit{inl}\ x' \Rightarrow e_1 \parallel \mathit{inr}\ y' \Rightarrow e_2) = e_2[y' \mapsto y] \\ \forall\, z : a + b.\, (\exists\, x : a.\, z = \mathit{inl}\ x) \vee (\exists\, y : b.\, z = \mathit{inr}\ y)\end{array}$$

Lifted types have constructors $\bot$ and $\mathit{lift} : \forall\, a.\, a \to a_\bot$:

$$\begin{array}{l}(\mathbf{case}\ \bot\ \mathbf{of}\ \mathit{lift}\ y \Rightarrow e) = \bot \\ (\mathbf{case}\ \mathit{lift}\ x\ \mathbf{of}\ \mathit{lift}\ y \Rightarrow e) = e[y \mapsto x] \\ \forall\, z : a_\bot.\, (z = \bot) \vee (\exists\, x : a.\, z = \mathit{lift}\ x)\end{array}$$

Finally, we assume β- and η-equivalence for functions, and that recursive definitions support unfolding and Scott induction.

Some abbreviations will be convenient. We write $\mathbf{let}\ x \triangleq e\ \mathbf{in}\ e'$ for $(\lambda\, x.\, e')\ e$, and $f \circ g$ for $\lambda\, x.\, f\ (g\ x)$. There are operators on functions corresponding to type constructors:

$$\begin{aligned} f_1 \times f_2 &= \lambda\, z.\, (f_1\ (\mathit{fst}\ z), f_2\ (\mathit{snd}\ z)) \\ f_1 + f_2 &= \lambda\, z.\, \mathbf{case}\ z\ \mathbf{of}\ \mathit{inl}\ x_1 \Rightarrow \mathit{inl}\ (f_1\ x_1) \parallel \mathit{inr}\ x_2 \Rightarrow \mathit{inr}\ (f_2\ x_2) \\ f_\bot &= \lambda\, z.\, \mathbf{case}\ z\ \mathbf{of}\ \mathit{lift}\ x \Rightarrow \mathit{lift}\ (f\ x) \end{aligned}$$

These operators preserve *id* and composition, i.e. they are functors.

3 A First-order Strict Language

The target language of the strictness translation is a first-order strict functional language, given in Fig. 3. The domain types of this language are identical to those of the lazy language, but there are no pointedness restrictions. Lifted types $t_\bot$

$$
\begin{array}{lll}
\text{Definitions} & def = p_d \triangleq \lambda\, x.\, v & \\
\text{Expressions} & v = x \quad | \quad \bot \quad | \quad () \quad | \quad (v_1, v_2) \quad | \quad p\ v & \\
& \quad |\ \textbf{case}\ v\ \textbf{of}\ \mathit{inl}\ x_1 \Rightarrow v_1 \ [\!]\ \mathit{inr}\ x_2 \Rightarrow v_2 & \\
& \quad |\ \textbf{delay}\ v & \\
\text{Partial} & p = p_d & \\
\text{functions} & \quad |\ \lambda\, x.\, v & \\
& \quad |\ \mathit{fst} \quad | \quad \mathit{snd} \quad | \quad \mathit{inl} \quad | \quad \mathit{inr} \quad | \quad \mathit{id} \quad | \quad \mathit{force} &
\end{array}
$$

Fig. 3. First-order strict functional language

$$
\left[\!\!\left[p_d \triangleq \lambda\, x.\, v \right]\!\!\right] = p_d \triangleq [\![\lambda\, x.\, v]\!]
$$

$$
\begin{aligned}
[\![p_d]\!] &= p_d \\
[\![\lambda\, x.\, v]\!] &= \lambda\, x.\ [\![v]\!] \\
[\![\mathit{fst}]\!] &= \mathit{lift} \circ \mathit{fst} \\
[\![\mathit{snd}]\!] &= \mathit{lift} \circ \mathit{snd} \\
[\![\mathit{inl}]\!] &= \mathit{lift} \circ \mathit{inl} \\
[\![\mathit{inr}]\!] &= \mathit{lift} \circ \mathit{inr} \\
[\![\mathit{force}]\!] &= \mathit{id} \\
[\![\mathit{id}]\!] &= \mathit{lift}
\end{aligned}
$$

$$
\begin{aligned}
[\![x]\!] &= \mathit{lift}\ x \\
[\![\bot]\!] &= \bot \\
[\![()]\!] &= \mathit{lift}\ () \\
[\![(v_1, v_2)]\!] &= \textbf{case}\ [\![v_1]\!]\ \textbf{of}\ \mathit{lift}\ x_1 \Rightarrow \\
&\quad\ \textbf{case}\ [\![v_2]\!]\ \textbf{of}\ \mathit{lift}\ x_2 \Rightarrow \\
&\quad\ \mathit{lift}\ (x_1, x_2) \\
[\![p\ v]\!] &= \textbf{case}\ [\![v]\!]\ \textbf{of}\ \mathit{lift}\ x \Rightarrow [\![p]\!]\ x \\
[\![\textbf{case}\ v\ \textbf{of}\ \mathit{inl}\ x_1 \Rightarrow v_1 \ [\!]\ \mathit{inr}\ x_2 \Rightarrow v_2]\!] & \\
&= \textbf{case}\ [\![v]\!]\ \textbf{of}\ \mathit{lift}\ y \Rightarrow \\
&\quad\ \textbf{case}\ y\ \textbf{of}\ \mathit{inl}\ x_1 \Rightarrow [\![v_1]\!]\ [\!] \\
&\qquad\qquad\quad \mathit{inr}\ x_2 \Rightarrow [\![v_2]\!] \\
[\![\textbf{delay}\ v]\!] &= \mathit{lift}\ ([\![v]\!])
\end{aligned}
$$

Fig. 4. Semantics of the strict language

have a special role in this language, representing the types of closures. These are built by the **delay** construct, and their values demanded by the function $\mathit{force} : \forall\, a.\, a_\bot \to a$, as in Scheme [5].

The semantics of this language is given in Fig. 4 via a translation into the lazy language, corresponding to the standard denotational semantics of strict languages. Strict computations of type t are modelled by lazy expressions of type $t_\bot$. Thus partial functions, whose types we write as $t \circ\!\!\rightarrow t'$, are translated to lazy functions of type $t \to {t'}_\bot$.

In the strict language we use abbreviations like those for the lazy language. We write **let** $x \triangleq v$ **in** v' for $(\lambda\, x.\, v')\ v$, and $p \circ q$ for $\lambda\, x.\, p\ (q\ x)$. The strict language also has operators (functors) preserving id and composition:

$$
\begin{aligned}
p_1 \times p_2 &= \lambda\, z.\, (p_1\ (\mathit{fst}\ z), p_2\ (\mathit{snd}\ z)) \\
p_1 + p_2 &= \lambda\, z.\, \textbf{case}\ z\ \textbf{of}\ \mathit{inl}\ x_1 \Rightarrow \mathit{inl}\ (p_1\ x_1)\ [\!]\ \mathit{inr}\ x_2 \Rightarrow \mathit{inr}\ (p_2\ x_2) \\
p_\bot &= \lambda\, z.\, \textbf{delay}\ (p\ (\mathit{force}\ z))
\end{aligned}
$$

$$
\begin{array}{rlllrcl}
\mathsf{Ide} & = \langle id & , & id & \rangle : & a & \trianglelefteq & a \\
\mathsf{Abs} & = \langle \lambda\,().\,\bot & , & \lambda\,x.\,() & \rangle : & \mathbb{1} & \trianglelefteq & a \\
\mathsf{Fail} & = \langle empty & , & \lambda\,x.\,\bot & \rangle : & \varnothing & \trianglelefteq & a \\
\alpha \times \beta & = \langle \alpha^{\mathrm{E}} \times \beta^{\mathrm{E}}, & & \alpha^{\mathrm{P}} \times \beta^{\mathrm{P}} & \rangle : & t'_\alpha \times t'_\beta & \trianglelefteq & t_\alpha \times t_\beta \\
\alpha + \beta & = \langle \alpha^{\mathrm{E}} + \beta^{\mathrm{E}}, & & \alpha^{\mathrm{P}} + \beta^{\mathrm{P}} & \rangle : & t'_\alpha + t'_\beta & \trianglelefteq & t_\alpha + t_\beta \\
\alpha_\bot & = \langle {\alpha^{\mathrm{E}}}_\bot & , & {\alpha^{\mathrm{P}}}_\bot & \rangle : & t'_{\alpha\bot} & \trianglelefteq & t_{\alpha\bot} \\
\alpha_\oplus & = \langle lift \circ \alpha^{\mathrm{E}}, & & \alpha^{\mathrm{P}} \circ force & \rangle : & t'_\alpha & \trianglelefteq & t_{\alpha\bot}
\end{array}
$$

Fig. 5. Embedding-projection pairs (given $\alpha : t'_\alpha \trianglelefteq t_\alpha$ and $\beta : t'_\beta \trianglelefteq t_\beta$)

4 Embedding-projection Pairs

We need to analyse each projection $\alpha : t \hookrightarrow t$ used in strictness analysis [13] as an e-p pair $\langle \alpha^{\mathrm{E}} : t' \to t, \alpha^{\mathrm{P}} : t \hookrightarrow t' \rangle$ such that $\alpha = {\alpha^{\mathrm{E}}}_\bot \circ [\![\alpha^{\mathrm{P}}]\!]$. That is, the embedding component is a total function, described in the language of lazy functions, while the projection component is a partial function, described in the language of strict functions. We shall identify α with this pair relating a type t with a reduced representation t'. We write $\alpha : t' \trianglelefteq t$ for this factorization.

The primitive e-p pairs and constructors are listed in Fig. 5.

The trivial e-p pair is the identity Ide, which embeds any type in itself.

The context Abs (absent) is used to indicate that a value is unused, so any value will do, and we may as well implement it using the unit type $\mathbb{1}$ (a form of dead code elimination). Note that $\mathsf{Abs}^{\mathrm{E}}$ is only defined for pointed types, while $\mathsf{Abs}^{\mathrm{P}}$ is defined for all types.

The context Fail is used to indicate that no value is acceptable. Such a value may be represented using the empty type.

We define the composition of e-p pairs $\alpha : t' \trianglelefteq t$ and $\beta : t'' \trianglelefteq t'$ as

$$\alpha \circ \beta \ = \langle \alpha^{\mathrm{E}} \circ \beta^{\mathrm{E}},\ \beta^{\mathrm{P}} \circ \alpha^{\mathrm{P}} \rangle :\ t'' \ \trianglelefteq \ t$$

The above e-p pairs satisfy the following equations:

$$\alpha = \alpha \circ \mathsf{Ide} = \mathsf{Ide} \circ \alpha \tag{5}$$

$$\mathsf{Abs} = \alpha \circ \mathsf{Abs} \qquad \text{if } [\![\alpha^{\mathrm{P}}]\!] \text{ is total} \tag{6}$$

$$\mathsf{Fail} = \alpha \circ \mathsf{Fail} \tag{7}$$

Next are the operators corresponding to the type constructors $\times$, $+$ (unlifted sum) and $\cdot_\bot$. By construction, these operations on e-p pairs preserve Ide and composition.

Liftings are eliminated (performing unboxing) by the variant $\cdot_\oplus$ of $\cdot_\bot$. This operator is not functorial; its action on Ide is the strictness projection:

$$\mathsf{Str} = \mathsf{Ide}_\oplus = \langle lift, force \rangle$$

The e-p pair corresponding to the projection Both is $\mathsf{Str} \times \mathsf{Str}$.

The interaction of the $\cdot_{\oplus}$ operator with composition is given by the following equations:

$$(\alpha \circ \beta)_{\oplus} = \alpha_{\oplus} \circ \beta \tag{8}$$
$$(\alpha \circ \beta)_{\oplus} = \alpha_{\perp} \circ \beta_{\oplus} \tag{9}$$

For example, the operation on e-p pairs corresponding to List is

$$\mathsf{List}\ \alpha \triangleq (\mathbb{1} + \alpha \times \mathsf{List}\ \alpha)_{\perp}$$

Often we will want to talk about lists in a strict context, so we use the active variant

$$\mathsf{List!}\ \alpha \triangleq (\mathbb{1} + \alpha \times \mathsf{List}\ \alpha)_{\oplus}$$

It would also be possible to eliminate the liftings in the recursive types, giving a context describing tail (or spine) strictness:

$$\mathsf{Tail}\ \alpha \triangleq (\mathbb{1} + \alpha \times \mathsf{Tail}\ \alpha)_{\oplus}$$

However, this kind of unboxing is not always a good idea, as the early evaluation of the whole list may use a great deal of space.

In order to be able to use the fixpoint methods developed for strictness analysis, we need to limit ourselves to a finite set of contexts. In the rest of this paper, we shall use contexts built from Abs, Fail and variables ranging over contexts using $\times$, $+$, $\cdot_{\perp}$ and $\cdot_{\oplus}$. We shall only unfold recursive contexts to one level, as we did with List above. We shall also introduce flat data types, like Num, whose associated contexts are instances of Ide.

4.1 Upper Bounds of Projections

There is a natural ordering on contexts, extending the ordering on simple projections: we write $\eta : \alpha \sqsubseteq \beta$ for $\alpha = \beta \circ \eta$. When we do not need to refer to η, we simply write $\alpha \sqsubseteq \beta$. A smaller e-p pair in this ordering represents less boxing, and is therefore more desirable.

In some of the rules of projection analysis, we require a context β greater than each of a pair of argument contexts α_1 and α_2. The actions on projections are the usual approximate answers [13] built from the primitive contexts and operators of Fig. 5. However, we also require e-p pairs $\eta_i : \alpha_i \sqsubseteq \beta$ to act as witnesses to the orderings, and to mediate between the representations. Since the approximations are defined by syntactic matching of the context terms, the necessary e-p pairs follow from various equations above, as in Fig. 6.

5 Translation from the Lazy Language

The translation of a lazy function f in a context β will be defined by a theorem of the form

$$[\![\beta^{\mathrm{P}}]\!] \circ f = [\![p \circ \alpha^{\mathrm{P}}]\!]$$

$$
\begin{aligned}
& \mathsf{Ide} : \alpha \sqsubseteq \alpha \\
& \alpha : \alpha \sqsubseteq \mathsf{Ide} \\
[\![\alpha^{\mathrm{P}}]\!] \text{ is total} \Rightarrow\ & \mathsf{Abs} : \mathsf{Abs} \sqsubseteq \alpha \\
& \mathsf{Fail} : \mathsf{Fail} \sqsubseteq \alpha \\
\eta_1 : \alpha_1 \sqsubseteq \beta_1 \wedge \eta_2 : \alpha_2 \sqsubseteq \beta_2 \Rightarrow\ & \eta_1 \times \eta_2 : \alpha_1 \times \alpha_2 \sqsubseteq \beta_1 \times \beta_2 \\
\eta_1 : \alpha_1 \sqsubseteq \beta_1 \wedge \eta_2 : \alpha_2 \sqsubseteq \beta_2 \Rightarrow\ & \eta_1 + \eta_2 : \alpha_1 + \alpha_2 \sqsubseteq \beta_1 + \beta_2 \\
\eta : \alpha \sqsubseteq \beta \Rightarrow\ & \eta_\perp : \alpha_\perp \sqsubseteq \beta_\perp \\
\eta : \alpha \sqsubseteq \beta \Rightarrow\ & \eta : \alpha_\oplus \sqsubseteq \beta_\oplus \\
\eta : \alpha \sqsubseteq \beta \Rightarrow\ & \eta_\oplus : \alpha_\oplus \sqsubseteq \beta_\perp
\end{aligned}
$$

Fig. 6. Rules for context orderings

where p is a partial function in the strict language and α is an e-p pair.

The simplest case, corresponding to the transformation discussed by Peyton Jones and Launchbury [19], is where β is Str. This corresponds to the common observation that "an expression is never evaluated unless its value is required in WHNF". Since $[\![\mathsf{Str}^{\mathrm{P}}]\!] = id$, the left side of the above equation reduces to f. In their terminology, the function f is defined as a *wrapper* performing unboxing of the arguments (via the projection α^{P}) for a *worker* function p. For example, we assume that the multiplication function is supplied with a translation

$$[\![\mathsf{Num}_\oplus{}^{\mathrm{P}}]\!] \circ (*) = \left[\!\left[(*) \circ (\mathsf{Num}_\oplus \times \mathsf{Num}_\oplus)^{\mathrm{P}}\right]\!\right]$$

stating that it is strict.

Often the context will be more interesting—in the above example the argument of f will be evaluated in the context α. The environment of the program may also imply a more informative context. For example, in a stream-producing computation the outer context will be both strict and head-strict.

The lazy functions of Fig. 2 interact in interesting ways with latent demand. In different contexts, *products* translates to different partial functions, both defined in Fig. 7:

$$
\begin{aligned}
\left[\!\left[(\mathsf{List!}\ \mathsf{Num}_\perp)^{\mathrm{P}}\right]\!\right] \circ \mathit{products} &= \left[\!\left[\mathit{products}_1 \circ (\mathsf{Num}_\perp \times \mathsf{List!}\ \mathsf{Num}_\perp)^{\mathrm{P}}\right]\!\right] \\
\left[\!\left[(\mathsf{List!}\ \mathsf{Num}_\oplus)^{\mathrm{P}}\right]\!\right] \circ \mathit{products} &= \left[\!\left[\mathit{products}_2 \circ (\mathsf{Num}_\oplus \times \mathsf{List!}\ \mathsf{Num}_\oplus)^{\mathrm{P}}\right]\!\right]
\end{aligned}
$$

In the first case, *products* is used in a completely lazy context, and the resulting program must build closures for both x' and the tail of the list, just like the original. In a head-strict context, *products* is compiled to a different version that performs the multiplication immediately, but still builds a closure for the tail. Further, $\mathit{products}_2$ expects as arguments an unboxed number and a list of unboxed numbers.

In contrast, a single translation of *append* suffices for all contexts List! α:

$$\left[\!\left[(\mathsf{List!}\ \alpha)^{\mathrm{P}}\right]\!\right] \circ \mathit{append} = \left[\!\left[\mathit{append}_1 \circ (\mathsf{List!}\ \alpha \times \mathsf{List}\ \alpha)^{\mathrm{P}}\right]\!\right]$$

$$
\begin{array}{l}
\textbf{type}\ \mathsf{List!}\ a \triangleq \mathbb{1} + a \times \mathsf{List}\ a \\[1em]
\mathit{products}_1 : \mathsf{Num}_\perp \times \mathsf{List!}\ \mathsf{Num}_\perp \multimap \mathsf{List!}\ \mathsf{Num}_\perp \\
\mathit{products}_1\ (x, n) \triangleq \textbf{case}\ n\ \textbf{of}\ \mathit{inl}\ () \Rightarrow \mathit{inl}\ ()\ \talloblong \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{inr}\ (y, \mathit{ys}) \Rightarrow \textbf{let}\ x' \triangleq \textbf{delay}\ (\mathit{force}\ x * \mathit{force}\ y)\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{inr}\ (x', \textbf{delay}\ (\mathit{products}_1\ (x', \mathit{force}\ \mathit{ys}))) \\[1em]
\mathit{products}_2 : \mathsf{Num} \times \mathsf{List!}\ \mathsf{Num} \multimap \mathsf{List!}\ \mathsf{Num} \\
\mathit{products}_2\ (x, n) \triangleq \textbf{case}\ n\ \textbf{of}\ \mathit{inl}\ () \Rightarrow \mathit{inl}\ ()\ \talloblong \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{inr}\ (y, \mathit{ys}) \Rightarrow \textbf{let}\ x' \triangleq x * y\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{inr}\ (x', \textbf{delay}\ (\mathit{products}_2\ (x', \mathit{force}\ \mathit{ys}))) \\[1em]
\mathit{append}_1 : \forall a.\ \mathsf{List!}\ a \times \mathsf{List}\ a \multimap \mathsf{List!}\ a \\
\mathit{append}_1\ (n, \mathit{ys}) \triangleq \textbf{case}\ n\ \textbf{of}\ \mathit{inl}\ () \Rightarrow \mathit{force}\ \mathit{ys}\ \talloblong \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{inr}\ (x, \mathit{xs}) \Rightarrow \mathit{inr}\ (x, \textbf{delay}\ (\mathit{append}_1\ (\mathit{force}\ \mathit{xs}, \mathit{ys})))
\end{array}
$$

Fig. 7. Partial functions generated by the translation

Note that append_1 is essentially the same as *append*—the unboxing projections are pushed straight through it onto its arguments.

Similar translations were achieved by the informally verified algorithm of Hall and Wise [10]. They use a single data type, which would be written in the notation of this paper as

$$\mathsf{Structure} \triangleq \mu\, c.\, (\mathsf{Atom} + c \times c)_\perp$$

where Atom is an unpointed set (including *nil* and numbers). Strictness is described by *strictness patterns*, which are equivalent (though with the inverse ordering) to certain contexts over Structure. Their outer strictness pattern corresponds to

$$\mathsf{Printer!} \triangleq (\mathsf{Atom} + \mathsf{Printer!} \times \mathsf{Printer})_\oplus$$

$$\mathsf{Printer} \triangleq (\mathsf{Atom} + \mathsf{Printer!} \times \mathsf{Printer})_\perp$$

They use an infinite domain of contexts, requiring the user to limit the number of versions of a function that may be generated.

5.1 Translation Rules

The rules of the backward analysis correspond to the rules of the conventional analysis [20,13]; what is new here is the translations.

The translations of the primitive functions are given in Fig. 8. These theorems follow from the axioms of Sect. 2, but the proofs are omitted here. Note that *lift* may also occur in a context of the form $\alpha_\perp$; this case is covered by rule (20), to be discussed later.

$$
\begin{aligned}
[\![\alpha^{\mathrm{P}}]\!] \circ \mathit{fst} &= \left[\!\left[\mathit{fst} \circ (\alpha \times \mathsf{Abs})^{\mathrm{P}}\right]\!\right] \\
[\![\alpha^{\mathrm{P}}]\!] \circ \mathit{snd} &= \left[\!\left[\mathit{snd} \circ (\mathsf{Abs} \times \alpha)^{\mathrm{P}}\right]\!\right] \\
\left[\!\left[(\alpha + \beta)^{\mathrm{P}}\right]\!\right] \circ \mathit{inl} &= [\![\mathit{inl} \circ \alpha^{\mathrm{P}}]\!] \\
\left[\!\left[(\alpha + \beta)^{\mathrm{P}}\right]\!\right] \circ \mathit{inr} &= [\![\mathit{inr} \circ \beta^{\mathrm{P}}]\!]
\end{aligned}
\qquad
\begin{aligned}
[\![\alpha_{\oplus}{}^{\mathrm{P}}]\!] \circ \mathit{lift} &= [\![\mathit{id} \circ \alpha^{\mathrm{P}}]\!] \\
[\![\alpha^{\mathrm{P}}]\!] \circ \mathit{id} &= [\![\mathit{id} \circ \alpha^{\mathrm{P}}]\!] \\
[\![\alpha^{\mathrm{P}}]\!] \circ \mathit{empty} &= [\![\mathit{id} \circ \mathsf{Fail}^{\mathrm{P}}]\!]
\end{aligned}
$$

Fig. 8. Translations of pre-defined functions

The translation of an expression e in a context β is described by a theorem of the form

$$[\![\beta^{\mathrm{P}}]\!] \;\; e = [\![v \lhd \rho^{\mathrm{P}}]\!]$$

for some translated expression v and environment ρ. The $\lhd$ operator, applied to an environment π mapping variables $x : t$ to call-by-value functions $\pi_x : t' \multimap t$, is the following abbreviation:

$$v \lhd \pi = \mathbf{let}\ x_1 \triangleq \pi_{x_1}\ x_1\ \mathbf{in}\ \ldots\ \mathbf{let}\ x_n \triangleq \pi_{x_n}\ x_n\ \mathbf{in}\ v$$

We write $\overline{\alpha}$ for the context environment mapping each variable to the context α.

Figure 9 contains inference rules for every combination of context and expression permitted by the type constraints of our language. The soundness of these follows from the axioms of Sect. 2, but the rules may also be read as a directed logic program.

The first two rules should be given precedence over the others, as they may produce superior results. First, a context of Abs indicates that any value will do, so we use the unit type instead. A context of Fail indicates that no value will do, so the empty type is a suitable representation.

The environment of variable values may be viewed as a product, and variables as projections. Hence their translation (12) is similar to *fst* and *snd* in Fig. 8.

If $\perp$ occurs in a strict context, no value for the free variables will do.

To compile a pair, we must combine the demands of the two components. On each variable, we will have a pair of e-p pairs $\alpha : t_\alpha \trianglelefteq t$ and $\beta : t_\beta \trianglelefteq t$. We require an e-p pair $\gamma : t_\gamma \trianglelefteq t$ and call-by-value functions $\theta^{\mathrm{L}} : t_\gamma \multimap t_\alpha$ and $\theta^{\mathrm{R}} : t_\gamma \multimap t_\beta$ such that

$$(\alpha^{\mathrm{P}}\ x, \beta^{\mathrm{P}}\ x) = \mathbf{let}\ x' \triangleq \gamma^{\mathrm{P}}\ x\ \mathbf{in}\ (\theta^{\mathrm{L}}\ x', \theta^{\mathrm{R}}\ x')$$

When this equation holds, we write $\langle \gamma, \theta^{\mathrm{L}}, \theta^{\mathrm{R}} \rangle \in \alpha \mathbin{\&} \beta$. As with the upper bound calculations, we can use the approximations for γ supplied by conventional projection analysis, but we also need the θ's. In Fig. 10, we go through all the rules for & [13] and calculate the θ's. Note the fall-back rule; the other rules give better answers where they apply, but this covers cases like $\alpha_\perp \mathbin{\&} \beta_\perp$. Recursive types can be handled in the same way as for ordinary projections [13].

Applications of primitive or user-defined functions are translated using (16). Applications of λ-expressions, i.e. **let** definitions, are translated using (17). The latter uses & because a **let** definition makes non-linear use of its environment, like the pairing construct.

$$\llbracket \mathsf{Abs}^{\mathrm{P}} \rrbracket \;\; e = \left\llbracket () \lhd \overline{\mathsf{Abs}}^{\mathrm{P}} \right\rrbracket \tag{10}$$

$$\llbracket \mathsf{Fail}^{\mathrm{P}} \rrbracket \;\; e = \left\llbracket \bot \lhd \overline{\mathsf{Fail}}^{\mathrm{P}} \right\rrbracket \tag{11}$$

$$\llbracket \beta^{\mathrm{P}} \rrbracket \;\; x = \left\llbracket x \lhd (\overline{\mathsf{Abs}}[x \mapsto \beta])^{\mathrm{P}} \right\rrbracket \tag{12}$$

$$\llbracket \beta^{\mathrm{P}} \rrbracket \;\; \bot = \left\llbracket \bot \lhd \overline{\mathsf{Fail}}^{\mathrm{P}} \right\rrbracket \qquad \llbracket \beta^{\mathrm{P}} \rrbracket \text{ strict} \tag{13}$$

$$\llbracket \beta^{\mathrm{P}} \rrbracket \;\; () = \left\llbracket () \lhd \overline{\mathsf{Abs}}^{\mathrm{P}} \right\rrbracket \tag{14}$$

$$\frac{\llbracket {\beta_i}^{\mathrm{P}} \rrbracket \;\; e_i = \llbracket v_i \lhd {\rho_i}^{\mathrm{P}} \rrbracket \qquad \langle \rho, \theta^{\mathrm{L}}, \theta^{\mathrm{R}} \rangle \in \rho_1 \,\&\, \rho_2}{\llbracket \beta_1 \times {\beta_2}^{\mathrm{P}} \rrbracket \;\; (e_1, e_2) = \llbracket (v_1 \lhd \theta^{\mathrm{L}}, v_2 \lhd \theta^{\mathrm{R}}) \lhd \rho^{\mathrm{P}} \rrbracket} \tag{15}$$

$$\frac{\llbracket \beta^{\mathrm{P}} \rrbracket \circ f = \llbracket p \circ \alpha^{\mathrm{P}} \rrbracket \qquad \llbracket \alpha^{\mathrm{P}} \rrbracket \;\; e = \llbracket v \lhd \rho^{\mathrm{P}} \rrbracket}{\llbracket \beta^{\mathrm{P}} \rrbracket \;\; (f \;\, e) = \llbracket (p \;\, v) \lhd \rho^{\mathrm{P}} \rrbracket} \tag{16}$$

$$\frac{\llbracket \beta^{\mathrm{P}} \rrbracket \;\; e_2 = \llbracket v_2 \lhd {\rho_2}^{\mathrm{P}} \rrbracket \qquad \llbracket {\rho_2}_x{}^{\mathrm{P}} \rrbracket \;\; e_1 = \llbracket v_1 \lhd {\rho_1}^{\mathrm{P}} \rrbracket \qquad \langle \rho, \theta^{\mathrm{L}}, \theta^{\mathrm{R}} \rangle \in \rho_1 \,\&(\rho_2 \backslash x)}{\llbracket \beta^{\mathrm{P}} \rrbracket \;\; \left(\mathbf{let}\ x \triangleq e_1 \ \mathbf{in}\ e_2\right) = \left\llbracket \left(\mathbf{let}\ x \triangleq v_1 \lhd \theta^{\mathrm{L}} \ \mathbf{in}\ v_2 \lhd \theta^{\mathrm{R}}\right) \lhd \rho^{\mathrm{P}} \right\rrbracket} \tag{17}$$

$$\frac{\llbracket \beta^{\mathrm{P}} \rrbracket \;\; e_i = \llbracket v_i \lhd {\rho_i}^{\mathrm{P}} \rrbracket \qquad \alpha_i = {\rho_i}_{y_i} \qquad \eta_i : \rho_i \backslash y_i \sqsubseteq \rho}{\llbracket \beta^{\mathrm{P}} \rrbracket \;\; \begin{pmatrix} \mathbf{case}\ x\ \mathbf{of} \\ \mathit{inl}\ y_1 \Rightarrow e_1 \parallel \\ \mathit{inr}\ y_2 \Rightarrow e_2 \end{pmatrix} = \left\llbracket \begin{pmatrix} \mathbf{case}\ x\ \mathbf{of} \\ \mathit{inl}\ y_1 \Rightarrow v_1 \lhd {\eta_1}^{\mathrm{P}} \parallel \\ \mathit{inr}\ y_2 \Rightarrow v_2 \lhd {\eta_2}^{\mathrm{P}} \end{pmatrix} \lhd (\rho[x \mapsto \alpha_1 + \alpha_2])^{\mathrm{P}} \right\rrbracket} \quad x \text{ not free in } e_1,\, e_2 \tag{18}$$

$$\frac{\llbracket \beta^{\mathrm{P}} \rrbracket \;\; e = \llbracket v \lhd \rho^{\mathrm{P}} \rrbracket \qquad \alpha = \rho_y}{\llbracket \beta^{\mathrm{P}} \rrbracket \;\; \left(\mathbf{case}\ x\ \mathbf{of}\ \mathit{lift}\ y \Rightarrow e\right) = \left\llbracket v \lhd ((\rho \backslash y)[x \mapsto \alpha_{\oplus}])^{\mathrm{P}} \right\rrbracket} \qquad \begin{array}{l} \llbracket \beta^{\mathrm{P}} \rrbracket \text{ strict} \\ x \text{ not free in } e \end{array} \tag{19}$$

$$\frac{\llbracket {\beta_{\oplus}}^{\mathrm{P}} \rrbracket \;\; e = \llbracket v \lhd \rho^{\mathrm{P}} \rrbracket \qquad \eta : \rho \sqsubseteq \rho' \qquad \overline{\mathsf{Abs}} \sqsubseteq \rho'}{\llbracket {\beta_{\bot}}^{\mathrm{P}} \rrbracket \;\; e = \left\llbracket \mathbf{delay}\ (v \lhd \eta^{\mathrm{P}}) \lhd {\rho'}^{\mathrm{P}} \right\rrbracket} \tag{20}$$

Fig. 9. Translation rules for expressions

$$\langle \alpha, \mathit{id}, \mathit{id} \rangle \in \alpha \,\&\, \alpha \qquad \langle \alpha, \mathit{id}, \lambda\, x.\, () \rangle \in \alpha \,\&\, \mathsf{Abs} \qquad \langle \mathsf{Fail}, \lambda\, x.\, \bot, \lambda\, x.\, \bot \rangle \in \alpha \,\&\, \mathsf{Fail}$$

$$\frac{\langle \gamma_1, \theta_1^{\mathrm{L}}, \theta_1^{\mathrm{R}} \rangle \in \alpha_1 \,\&\, \beta_1 \qquad \langle \gamma_2, \theta_2^{\mathrm{L}}, \theta_2^{\mathrm{R}} \rangle \in \alpha_2 \,\&\, \beta_2}{\langle \gamma_1 \times \gamma_2, \theta_1^{\mathrm{L}} \times \theta_2^{\mathrm{L}}, \theta_1^{\mathrm{R}} \times \theta_2^{\mathrm{R}} \rangle \in (\alpha_1 \times \alpha_2) \,\&(\beta_1 \times \beta_2)}$$

$$\frac{\langle \gamma_1, \theta_1^{\mathrm{L}}, \theta_1^{\mathrm{R}} \rangle \in \alpha_1 \,\&\, \beta_1 \qquad \langle \gamma_2, \theta_2^{\mathrm{L}}, \theta_2^{\mathrm{R}} \rangle \in \alpha_2 \,\&\, \beta_2}{\langle \gamma_1 + \gamma_2, \theta_1^{\mathrm{L}} + \theta_2^{\mathrm{L}}, \theta_1^{\mathrm{R}} + \theta_2^{\mathrm{R}} \rangle \in (\alpha_1 + \alpha_2) \,\&(\beta_1 + \beta_2)}$$

$$\frac{\langle \gamma, \theta^{\mathrm{L}}, \theta^{\mathrm{R}} \rangle \in \alpha \,\&\, \beta}{\langle \gamma_{\oplus}, \theta^{\mathrm{L}}, \theta^{\mathrm{R}} \rangle \in \alpha_{\oplus} \,\&\, \beta_{\oplus}} \qquad \frac{\alpha \sqsubseteq \gamma \qquad \eta : \beta \sqsubseteq \gamma \qquad \langle \delta, \theta^{\mathrm{L}}, \theta^{\mathrm{R}} \rangle \in \alpha \,\&\, \gamma}{\langle \delta_{\oplus}, \theta^{\mathrm{L}}, (\lambda\, z.\, \mathbf{delay}\ (\eta^{\mathrm{P}}\ z)) \circ \theta^{\mathrm{R}} \rangle \in \alpha_{\oplus} \,\&\, \beta_{\bot}}$$

$$\frac{\eta_1 : \alpha \sqsubseteq \gamma \qquad \eta_2 : \beta \sqsubseteq \gamma}{\langle \gamma, {\eta_1}^{\mathrm{P}}, {\eta_2}^{\mathrm{P}} \rangle \in \alpha \,\&\, \beta} \qquad \text{(used if no other rule applies)}$$

Fig. 10. Rules for the & operation

Equations (18) and (19) cover only restricted forms, where the expression being analysed is a variable that does not occur free in the body of the **case**. In this way we avoid aliasing, giving simpler rules. However, no generality is lost, as any **case** expression may be re-written in this form using a **let**.

5.2 Active and Latent Contexts

Equations (13) and (19) have an extra side condition: the function $[\![\beta^{\mathrm{P}}]\!]$ must be strict, corresponding to a context in which the value is demanded.

A similar restriction arises in the translation of recursive functions. A function $f \triangleq \lambda\, x.\, e$ will be translated to a different partial function p for each context β. Since our contexts correspond to those of ordinary projection analysis [20,13], we can use standard fixed point methods to compute the input context α for the required contexts β, and then read off the associated translation p for a particular β. Now the equations $[\![\beta^{\mathrm{P}}]\!] \circ f = [\![p \circ \alpha^{\mathrm{P}}]\!]$ are inductive, so they will hold for the fixed point functions if $[\![\beta^{\mathrm{P}}]\!]$ is strict.

For example, $\gamma_{\oplus}$ has a strict projection component, while $\gamma_{\perp}$ does not. Recall that the translation of *lift* in Fig. 8 also requires the strict form.

Wadler and Hughes [20] made all contexts strict by introducing a *guard rule*. We shall only do this where required by the above rules. They may be extended to contexts of the form $\beta_{\perp}$ using (20). That is, create a boxed closure, whose code evaluates v in an active context.

In most accounts of strictness analysis, all values are assumed to be of lifted type, so (20) covers all cases. In our lazy language, we require of the result type of the evaluation only that it be pointed. Hence it cannot be a sum, but it may be a product of pointed types. Now the context $\gamma \times \delta$ has a strict projection component if either of γ or δ have. Otherwise, in a context $\gamma \times \delta$ we can rewrite an evaluation using the equation

$$\begin{aligned} \textbf{case}\ x\ \textbf{of}\ \mathit{lift}\ y \Rightarrow e = \textbf{let}\ z &\triangleq (\textbf{case}\ x\ \textbf{of}\ \mathit{lift}\ y \Rightarrow \mathit{lift}\ e)\ \textbf{in} \\ &(\textbf{case}\ z\ \textbf{of}\ \mathit{lift}\ p \Rightarrow \mathit{fst}\ p, \\ &\ \textbf{case}\ z\ \textbf{of}\ \mathit{lift}\ p \Rightarrow \mathit{snd}\ p) \end{aligned} \tag{21}$$

In the expression on the right-hand side, the first **case** is of lifted type, and thus will have a tractable context. The other two **case** have contexts γ and δ, which we treat inductively. The generated code for such fully latent pairs will not be efficient, but the programmer has left us little choice.

A similar device may be applied to recursively defined functions.

5.3 Polymorphism

Since our language allows expressions of pointed types in these positions, the other possibility is that the result type of an evaluation or recursive function is a type variable, constrained to pointed types. Such a case must be treated by instantiation of the type variable, creating a context more specific than the type of the expression.

Instantiation is also required in the calculation of certain upper bounds of contexts. This is because Fig. 6 gives no general rule for an upper bound of α and Abs.

However, if we constrain the language so that the types of these expressions must be products of lifted types, then our translation procedure will be polymorphic. Even this restriction is less stringent than the situation in languages like Haskell [11], where all expressions are of lifted type.

6 Conclusion

We have developed an algorithm to compile lazy functions to call-by-value forms. By basing the algorithm on projection-based strictness analysis, we were able to make use of latent demand information in the translation. Moreover, the rules of our algorithm are formulas over a simple functional language, and are hence easily verified.

The key step was factoring of the projections as embedding-projection pairs. In the present paper, we used only the projection components in a backwards analysis to expose strictness and perform unboxing. It is also possible (indeed simpler) to define a forwards analysis using the embeddings, delaying creation of closures, possibly permanently. It is also possible that other analyses using projections may benefit from factorization.

The principal shortcoming of our algorithm is inherited from the analysis on which it is based: it is limited to first-order functions. This is an obvious area for future work, perhaps using one of the higher-order extensions of projection analysis [7,12].

Acknowledgements

John Launchbury suggested that an explicit treatment of lifting might enable the exploitation of projection-based strictness analysis. John Hughes, David Sands and anonymous referees gave suggestions for the presentation of this work. Nurith Goldblatt found some technical errors.

References

1. Torben Amtoft. Minimal thunkification. In Patrick Cousot et al., editors, *Static Analysis: 3rd International Workshop, WSA '93, Padova, Italy*, volume 724 of *Lecture Notes in Computer Science*, pages 218–229. Springer, 1993.
2. Andrew A. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.
3. Geoffrey L. Burn, Chris Hankin, and Samson Abramsky. Strictness analysis of higher order functions. *Science of Computer Programming*, 7:249–278, 1986.
4. Geoffrey L. Burn and Daniel Le Métayer. Proving the correctness of compiler optimisations based on a global program analysis. In *Fifth International Symposium on Programming Language Implementation and Logic Programming*, volume 714 of

Lecture Notes in Computer Science, pages 346–364, Tallinn, Estonia, August 1993. Springer.
5. W. Clinger and J. Rees. Revised^4 report on the algorithmic language Scheme. *ACM Lisp Pointers*, IV, July–September 1991.
6. Olivier Danvy and John Hatcliff. CPS-transformation after strictness analysis. *ACM Letters on Programming Languages and Systems*, 1(3):195–212, 1992.
7. Kei Davis and Philip Wadler. Strictness analysis in 4D. In *1990 Glasgow Workshop on Functional Programming*, Workshops in Computing, pages 23–43. Springer, 1991.
8. Karl-Filip Faxén. Optimizing lazy functional programs using flow inference. In *Static Analysis, Second International Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 136–153. Springer, 1995.
9. Karl-Filip Faxén. Flow inference, code generation and garbage collection for lazy functional languages. Licentiate thesis, Kungl Tekniska Högskolan, January 1996.
10. Cordelia V. Hall and David S. Wise. Compiling strictness into streams. In *14th ACM Symposium on Principles of Programming Languages*, pages 132–143, Munich, January 1987.
11. Paul Hudak, Simon Peyton Jones, Philip Wadler, et al. Report on the programming language Haskell, a non-strict purely functional language (Version 1.2). *SIGPLAN Notices*, 27(5), March 1992.
12. Sebastian Hunt. *Abstract Interpretation of Functional Languages: From Theory to Practice*. PhD thesis, Imperial College of Science, Technology and Medicine, 1991.
13. Ryszard Kubiak, John Hughes, and John Launchbury. Implementing projection-based strictness analysis. In *1991 Glasgow Workshop on Functional Programming*, Workshops in Computing, pages 207–224. Springer, 1992.
14. T-M. Kuo and P. Mishra. Strictness analysis: A new perspective based on type inference. In *FPCA'89, London, England*, pages 260–272. ACM Press, September 1989.
15. John Launchbury and Gebreselassie Baraki. Representing demand by partial projections. *Journal of Functional Programming*, 6(4), 1996.
16. John Launchbury and Ross Paterson. Parametricity and unboxing with unpointed types. In *European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 204–218, Linköping, Sweden, April 1996.
17. Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In B. Robinet, editor, *International Symposium on Programming*, volume 83 of *Lecture Notes in Computer Science*. Springer, 1980.
18. Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, July 1992.
19. Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Conference on Functional Programming Languages and Computer Architecture*, pages 636–666, Cambridge, MA, 1991.
20. Philip Wadler and John Hughes. Projections for strictness analysis. In *Conference on Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, Portland, OR, 1987.