J. Functional Programming **6** (4): 563–585, July 1996 © 1996 Cambridge University Press

Representing demand by partial projections

JOHN LAUNCHBURY AND GEBRESELASSIE BARAKI

Oregon Graduate Institute, and Glasgow University

(e-mail: `jl@cse.ogi.edu`*)*

Abstract

The projection-based strictness analysis of Wadler and Hughes is elegant and theoretically satisfying except in one respect: the need for lifting. The domains and functions over which the analysis is performed need to be transformed, leading to a less direct correspondence between analysis and program than might be hoped for. In this paper we shall see that the projection analysis may be reformulated in terms of partial projections, so removing this infelicity. There are additional benefits of the formulation: the two forms of information captured by the projection are distinguished, and the operational significance of the range of the projection fits exactly with the theory of unboxed types.

Capsule Review

This paper provides a reformulation of projection-based program analysis using partial projections. The advantage is that by working with partial projections the Hughes/Wadler 'lightening bolt' becomes unnecessary and, as a result, the theory is much more elegant. Projections-based analyses capture two types of demand: 'active' demands which require arguments of a specific form and 'latent' demands which do not require argument evaluation but cause failure when combined with an active demand. The current formulation highlights these two types of demand, whereas the earlier formulation (with lifting) tended to obscure the differences. The authors also show that there is a good match between the partial projections and the theory of unboxed types. The paper is concluded by a section relating partial projections to partial equivalence relations; the latter were used by Sebastian Hunt to provide a framework for abstract interpretation and Hughes/Wadler projections. The work based on partial equivalence relations treats higher-order features in a rather natural way; it will be interesting to see if the partial projections work can be extended to higher-order – the framework presented in this paper is likely to prove more accessible to a wider audience than the partial equivalence relations framework.

1 Introduction

The method of projection-based backwards strictness analysis for first-order, lazy functional languages was first presented by Wadler and Hughes (1987), and has undergone significant development since then. The method is elegant and theoretically satisfying except in one respect: the need for lifting. While projections are great at representing complex strictness patterns, they cannot represent simple strictness.

The solution adopted from early on is to give an alternative interpretation to programs in which all the domains are lifted with an additional bottom element. This adds complication and confusion, and provides far less of a direct correspondence between the analysis and the program's semantics than might be hoped for.

In this paper we propose an alternative technique for capturing simple strictness: using *partial projections*. While the change is small, its ramifications are significant. No longer is an alternative presentation of domains required, and no longer is an alternative program semantics required with all its potential for confusion. Instead, the analysis relates directly with the standard semantics, and the projections are defined structurally with respect to the standard formation of domains.

In addition, using partial projections has benefits with regard to operational intuition. The two roles of a projection, that of propagating demand, and that of equating distinct values, are distinguished: the former corresponding directly with the lower sets which arise in BHA strictness analysis (Burn *et al.*, 1986), and the latter with the equivalence classes of Hunt's PERs (Hunt, 1991). Furthermore, the link between projection analysis and *unboxed types* (Peyton Jones and Launchbury, 1991) becomes clearer, though the full implications remain to be studied.

Projection analysis has progressed significantly since the early days, yet this paper is fairly complete in its coverage of modern projection analysis. That is, despite the change to partial projections, all the development of the last few years carries through largely unchanged.

We begin with a brief review of the history and method of projection-based strictness analysis. Then we introduce partial functions and partial projections, after which we introduce our example language, with its model for types and its semantics. We provide an abstract semantics based on partial projections which defines a strictness analysis. We then prove the correctness of the analysis.

2 Historical background

Detecting strictness by backwards analysis was originally proposed by Johnsson (1985), who used a two point domain, then developed by Wray using a four point domain (Wray, 1985). However, backwards analysis came into its own under Hughes (1985), who demonstrated the power of propagating demands for finding strictness within data structures. He used *contexts* – evaluators defined over a universal domain representing closures (delayed evaluations) – and relied on performing complex series of simplifications to the contexts. Unfortunately, this caused the results of the analysis to degrade unpredictably. Moreover, the transformations of contexts received no more than informal justification. Hughes concluded that the work was "a demonstration of a possibility, and a potential basis for further work".

In 1987, two other papers were published which brought the work closer to practicality. The first introduced abstractions of Hughes' original context domains which were much more tractable, and demonstrated a number of other analyses which fitted the backwards mould (Hughes, 1987). In addition, he showed how the technique could be extended to higher-order functions, as a combination of abstract interpretation and backwards analysis.

The second paper, written in conjunction with Wadler, is now seen as the seminal paper on projection-based analysis (Wadler and Hughes, 1987). The thrust of the paper was that contexts could be modelled semantically by domain projections defined over domains representing individual data types. This was a significant development. It became far easier to perform proofs of correctness for the analysis, and it was now possible to define finite domains of projections which captured standard strictness patterns. There was a problem, however. While projections could capture rich strictness patterns within data structures, they were completely unable to capture simple strictness! The solution adopted in the paper was to lift the domains and introduce a new bottom element (called 'lightning bolt'). Then simple strictness could be represented. We discuss this device in more detail in section 3.1.

In the same year, Launchbury demonstrated that the same projection framework could be used in partial evaluation to express binding-time analysis (Launchbury, 1987), though here there was no need for lifting. Interestingly, this analysis was a forwards analysis, demonstrating that projections had no inherent direction of analysis. Indeed this was shown in later papers by Hughes and Launchbury where the correspondence between these forwards and backwards analyses was explored (Hughes and Launchbury, 1991; Launchbury, 1991b). Again, the presence of lifting in strictness analysis meant that the two analyses were hard to compare formally. They existed in different worlds, and the mapping between these worlds was non-trivial.

Since then, projection-based analysis has been generalised to handle polymorphism and arbitrary user-defined data types, though only for first order functions (Hughes and Launchbury, 1992), it has been implemented both for binding-time and strictness analysis (Launchbury, 1991a; Kubiak *et al.*, 1992), and further refined to handle higher-order functions (Davis, 1993), though non-polymorphically as yet.

In addition, the relationship between projection-based backwards analysis and BHA abstract interpretation (Burn *et al.*, 1986) has received some attention. Burn (1990) explored the correspondence between the Scott-closed sets of BHA and the so-called *lift-strict* projections, though once again the presence of lifting caused confusion: should (non-lifted) BHA be compared with the lifted or the non-lifted version of projection analysis?

More recently still, Hunt developed analyses based on partial equivalence relations (PERs). The analysis was applied both to strictness and to binding-time analysis (Hunt, 1991; Hunt and Sands, 1991). At the base types, the PERs used by Hunt corresponded exactly to the non-lifted projections, except for an anomolous one introduced especially to capture strictness. We shall discuss this more in section 6.2.

The question then is, can we do better? Can projection-based analysis (and even PER-analysis) be reformulated so that simple strictness does not need to be handled specially?

3 Intuition

Recall that, in domain theory, a projection is a function α such that $\alpha \circ \alpha = \alpha$ and $\alpha \sqsubseteq Ide$ $(= \lambda x.x)$. The essential intuition for strictness analysis is that a projection performs a certain amount of evaluation of a lazy data-structure. For example, the projection

$$\begin{aligned} &\mathit{Left}\ : \mathit{Nat} \times \mathit{Nat} \to \mathit{Nat} \times \mathit{Nat} \\ &\mathit{Left}\ (x, y) \ = \ (\bot, \bot) \quad \textbf{if } x = \bot \\ &\qquad\qquad\quad = \ (x, y) \quad \textbf{otherwise} \end{aligned}$$

may be thought of as evaluating the first component of a pair, while

$$\begin{aligned} &\mathit{Both}\ : \mathit{Nat} \times \mathit{Nat} \to \mathit{Nat} \times \mathit{Nat} \\ &\mathit{Both}\ (x, y) \ = \ (\bot, \bot) \quad \textbf{if } x = \bot \textbf{ or } y = \bot \\ &\qquad\qquad\quad = \ (x, y) \quad \textbf{otherwise} \end{aligned}$$

evaluates both. Now we can regard a function as *Both*-strict – performing as much evaluation as *Both* – if evaluating its argument with *Both* before the call does not change its result. For example, the function $+ : \mathit{Nat} \times \mathit{Nat} \to \mathit{Nat}$ evaluates both its arguments, and so $+ = + \circ \mathit{Both}$. More generally, there may be parts of a function's argument that are evaluated only if certain parts of its result are evaluated – a function may evaluate more or less of its argument depending on context. Take *swap* for example.

$$\begin{aligned} &\mathit{swap}\ : \mathit{Nat} \times \mathit{Nat} \to \mathit{Nat} \times \mathit{Nat} \\ &\mathit{swap}\ (x, y) = (y, x) \end{aligned}$$

While *swap* is not *Both*-strict, it is *Both*-strict in a *Both*-strict context since

$$\mathit{Both} \circ \mathit{swap} = \mathit{Both} \circ \mathit{swap} \circ \mathit{Both}$$

Thus, if both components of *swap*'s result will be evaluated, then the components of its argument can be evaluated before the call without changing the meaning. We make the following definition:

Definition

Let f be a function and α and β be projections. We say f is α-strict in a β-strict context if $\beta \circ f = \beta \circ f \circ \alpha$ (or equivalently, $\beta \circ f \sqsubseteq f \circ \alpha$).

The definition has a very natural reading as a means of propagating demand. If we demand β of $(f \circ g)$, say, then

$$\beta \circ (f \circ g) = \beta \circ f \circ \alpha \circ g$$

and so the demand of β on f has now been propagated to become a demand of α on g.

Using this formulation, the aim of strictness analysis may be summarised as follows. From f, together with a projection β representing the demand for the result of f, we want to find an α such that $\beta \circ f \sqsubseteq f \circ \alpha$. We can always choose α to be the identity projection *Ide*, but this is uninformative: *Ide* corresponds to performing no evaluation at all. We would like to find the smallest α such that the condition holds. In general this is equivalent to the halting problem, but methods are known for finding quite small αs, and later in this paper we give an example of these methods.

3.1 Lifting

In some sense projections capture the notion of evaluating a component *within* a data-structure. They cannot capture the notion of evaluating a single value. A trick suggested by Wadler and Hughes is always to embed simple values within a 'data-structure' having a single component, which we can think of as representing an unevaluated closure.

In Wadler and Hughes' framework, we think of a closure of type t as an element of $t_\perp$, and we 'evaluate' it with the projection

$$\begin{array}{llll} Str : t_\perp \rightarrow t_\perp & & & \\ Str \ \perp & = & \perp & \\ Str \ (lift \ x) & = & \perp & \textbf{if } x = \perp \\ & = & lift \ x & \textbf{otherwise} \end{array}$$

(writing the non-$\perp$ elements in the form $lift \ x$). Now, any function $f : s \rightarrow t$ induces a function $f_\perp : s_\perp \rightarrow t_\perp$ which behaves like f on elements of s, but maps the new $\perp$ to $\perp$. It is easy to show that,

$$f \text{ is strict} \Leftrightarrow Str \circ f_\perp \sqsubseteq f_\perp \circ Str$$

This is fine except that now, rather than analysing f, we end up analysing $f_\perp$. This has consequences for all the types of f's arguments which too must be lifted. So, for example, if f has the type

$$f : (Int \times Int) \rightarrow Int$$

then the analysis actually works with

$$f_\perp : (Int \times Int)_\perp \rightarrow Int_\perp$$

though, in turn, the lifting on f's argument needs to be pushed through the product using the isomorphism $(A \times B)_\perp \cong A_\perp \otimes B_\perp$ giving

$$f_\perp : Int_\perp \otimes Int_\perp \rightarrow Int_\perp$$

where $\otimes$ is smash product.

If this was the limit of the additional complexity then things wouldn't be too bad, but as was shown in Kubiak *et al.* (1992), this and other isomorphisms work their way into the definitions of structured domains as well. Consequently, it is not only the functions that become modified by lifting, but the very domains that they manipulate suffer too.

On the other hand, lifting has given exactly what we want. There are four fundamental projections over lifted types which capture various degrees of evaluation, and these correspond to Wray's original 4-point domain. We have already seen *Ide* (no evaluation), and *Str* (evaluate). In addition there is *Abs* defined by

$$\begin{array}{lll} Abs : t_\perp \rightarrow t_\perp & & \\ Abs \ \perp & = & \perp \\ Abs \ (lift \ x) & = & lift \ \perp \end{array}$$

and the constant bottom function *Fail* ($= \lambda x.\perp$). These are discussed in more detail later when they reappear as partial projections.

In the next section we see how broadening our view to include partial projections provides all the advantages of lifting without its ugliness.

4 Partial functions

Denotational semantics has traditionally been formulated in the category of complete (pointed) partial orders with continuous functions (which we write $\mathscr{C}$). Plotkin has reformulated it in the context of *partial continuous functions* (Plotkin, 1985). We call this category $\mathscr{P}$; objects of this category are *unpointed* complete partial orders (Schmidt, 1986) – they don't necessarily have a bottom element.

It is inappropriate here to cover the theory of partial functions in great depth. However, all the necessary intuition may be gained by recognising that $\mathscr{P}$ may be injected into the category $\mathscr{S}$ of pointed complete partial orders and *strict* continuous functions by lifting (defined slightly more generally than before). On cpo's, lifting adds a new bottom element (making the cpo pointed if it was not previously). On partial functions we have:

$$\begin{array}{lcll} f_\perp\ \perp & = & \perp & \\ f_\perp\ (\mathit{lift}\ x) & = & \perp, & \textbf{if } f\ x = \text{<UNDEF>} \\ & = & \mathit{lift}\ (f\ x), & \textbf{otherwise} \end{array}$$

where, as before, we write *lift* x for elements coming from the original domain.

The major insight gained from this transformation is that partial continuous functions may be undefined only on some lower portion of their source (i.e. a Scott-closed set). Undefinedness cannot occur at arbitrary points. By providing an extra bottom on the target domain the function may be made total, mapping to the new bottom where previously it was undefined.

Note that partial functions are 'strict' in undefinedness in the sense that if $g\ x$ is undefined, then so is $f\ (g\ x)$, whatever the definition of f. For this reason, $\mathscr{P}$ is ideal for modelling strict functional languages, taking non-termination to correspond to partiality. Similarly, partial functions correspond very naturally with low-level machine concepts where immediate (rather than delayed) evaluation is the norm. One may almost describe $\mathscr{P}$ as the category for implementation semantics, whereas $\mathscr{C}$ is more convenient for reasoning.

In $\mathscr{P}$ we have to use Kleene's equality: $e_1 = e_2$ means that either both e_1 and e_2 are undefined, or they are both defined and equal. Similarly, $e_1 \sqsubseteq e_2$ means that either e_1 is undefined, or they are both defined and e_1 is dominated by e_2 in the partial order of the cpo.

Not only is there an injection of $\mathscr{P}$ into $\mathscr{C}$, but $\mathscr{P}$ includes $\mathscr{C}$ as a sub-category, so we may view our language semantics as being within $\mathscr{P}$, but with no partiality, i.e. delays everywhere and no explicit strictness behaviour. Using partial projections we will introduce strictness behaviour, and propagate it through the program by the projection analysis. Thus the results of projection analysis may be viewed as improving upon the naive translation of $\mathscr{C}$ semantics into $\mathscr{P}$ semantics that is given by the natural embedding.

4.1 Partial projections

The injection from $\mathscr{P}$ into $\mathscr{S}$ was built into the original formulation of projection analysis (though it hasn't previously been perceived as such). All the projections which arise in the analysis, therefore, are images of projections which exist in $\mathscr{P}$. For this reason we shall reuse the names, and no longer think of them as being projections over lifted domains. By viewing them in $\mathscr{P}$ we find we can see the structure more clearly, without the lifting encoding obscuring the essential behaviour.

The four basic projections may be defined as follows:

$$\begin{aligned} Ide\ x &= x \\ Abs\ x &= \bot \\ Str\ x &= \text{<UNDEF>} \quad \textbf{if } x = \bot \\ &= x \quad\quad\quad\quad \textbf{otherwise} \\ Fail\ x &= \text{<UNDEF>} \end{aligned}$$

These definitions are simpler than their lifted images given earlier because we no longer need to code up the behaviour of the lifting transformation within the projections.

4.1.1 Demands

A partial projection corresponds to a demand: the values on which the projection is undefined are 'unacceptable'. Any other value is fine. So, for example, if a term is under evaluation and is needed in weak head normal form (i.e. the outermost constructor is required) then $\bot$ is unacceptable as a result, but any other value is fine. The demand on the value is expressed by *Str*.

Now if we define, `f x = bottom` and place a demand of *Str* on the result of f, what is the demand on f's argument? The answer is *Fail*. There is no value for x which makes $f\ x$ return a non-bottom value, so every value of x is unacceptable. Formally,

$$Str \circ f = Str \circ f \circ Fail$$

where f is the semantic image of `f`.

In contrast, if we define, `g x = 3` then a demand of *Str* on the result of g propagates to a demand of *Abs* on x. No value is unacceptable, and furthermore, all values are equivalent. Formally,

$$Str \circ g = Str \circ g \circ Abs$$

A projection may be undefined only on some lower portion of its domain, a Scott-closed set. For example, *Str* corresponds with $\{\bot\}$ as it is undefined only on $\bot$, whereas *Fail* corresponds with the whole domain. On the other hand, *Ide* corresponds to $\{\}$, as it is defined everywhere.

Truly-partial projections (i.e. projections which are undefined on a non-empty set of values) correspond to the projections that have previously been (improperly) called strict-projections (Wadler and Hughes, 1987) or, more properly, 'lift-strict' (Burn, 1990). However, (partial) projections go beyond this. The projection *Abs*, for example, implies no demand, but it states that as far as the computation is concerned, all values are equivalent. Thus, if ever any active demand (such as *Str*) were combined with *Abs*, the demand on the term would immediately become *Fail*: all values are equivalent, and at least one is unacceptable; therefore all values are unacceptable. It is possible to see the equivalence of values specified by *Abs* as a 'latent demand': if ever one value becomes unacceptable, then all the values equated by the projection also become unacceptable. Perhaps an analogy is in order. A projection corresponding to a latent demand is like a pilot with sealed orders. Sealed orders are merely *potential orders* – they may never happen. However, the order to open the envelope also activates the orders within.

The four projections we have been discussing exist over every domain, so they are necessarily limited. Domains with a richer structure possess richer projections. List domains, for example, possess a projection *Head* defined by,

$$\begin{array}{lcll} \mathit{Head}\ [\,] & = & [\,] & \\ \mathit{Head}\ (x : xs) & = & \bot, & \textbf{if } x = \bot \\ & & x : \mathit{Head}\ xs, & \textbf{otherwise} \end{array}$$

(where we write [] for nil, and : for cons). This *Head* projection contains lots of pockets of equivalences working along a list. For example, $\mathit{Head}\ (\bot : [\,]) = \bot$, or, $\mathit{Head}\ (2 : (\bot : (3 : [\,]))) = \mathit{Head}\ (2 : \bot)$. Each equivalence corresponds to a latent demand which, if any value in an equivalence class ever becomes unacceptable, then all the values in the same class become unacceptable also. This concept forms the essence of Hunt's PER-based approach to strictness analysis (Hunt, 1991).

In previous formulations of projection strictness analysis, the two forms of demand were confused (literally) by the lifting trick. Both were simply equivalences. Without the trick, their distinct nature becomes clear. Active demands are represented by undefinedness, latent demands by equivalences.

4.1.2 Combining projections

Wadler and Hughes introduced two operations for combining projections: $\sqcup$ and &. The first is usual least upper bound (i.e. pointwise). The second is similar, except that it conjoins demand. In the setting of partial projections they may be defined as follows.

$$\begin{array}{lcll} (\alpha \sqcup \beta)\ x & = & \beta\ x & \textbf{if } \alpha\ x = \text{<UNDEF>} \\ & = & \alpha\ x & \textbf{if } \beta\ x = \text{<UNDEF>} \\ & = & (\alpha\ x) \sqcup (\beta\ x) & \textbf{otherwise} \end{array}$$

$$\begin{array}{lcll} (\alpha \& \beta)\ x & = & \text{<UNDEF>} & \textbf{if } \alpha\ x = \text{<UNDEF>} \textbf{or } \beta\ x = \text{<UNDEF>} \\ & = & (\alpha\ x) \sqcup (\beta\ x) & \textbf{otherwise} \end{array}$$

In contrast with the earlier setting, both of these functions arise entirely naturally, again because of the clear distinction between active and latent demands.

Any truly-partial projection is expressible in the form $Str \circ \alpha$ where α can be total or partial. Similarly any total projection can be written in the form $Abs \sqcup \alpha$, where again α may be either total or partial. The first of these formulations is so important that we denote it with a special notation. We define, $!\alpha = Str \circ \alpha$.

5 A particular analysis

So far we have concentrated only on the broad aspects of basing a strictness analysis on partial projections, and have avoided being explicit about any details. Rather, what we have discussed applies to the whole field of projection-based strictness analysis. In this section we become more specific and demonstrate the ideas in practice by reworking the analysis of Kubiak *et al.* (1992).

We introduce an example language and provide it with a concrete semantics. Then we define the strictness analysis using partial projections and provide a proof of correctness. This is done in detail as some aspects of manipulating terms denoting partial functions are not always entirely familiar. Finally, the section ends with a number of examples of the sort of information that the analysis is able to discover.

5.1 Syntax

We use a polymorphic, first-order, lazy functional language with user-defined data types, whose syntax is similar to Haskell. Programs consist of type definitions and function definitions. For simplicity, we require each function definition to have explicit type information. An example program is

```
data List a = rec L . Nil | Cons a L

append :: List a -> List a -> List a
append xs ys
  = case xs of
      Nil -> ys
      Cons u us -> Cons u (append us ys)
```

The syntax is presented in figure 1. The grammar uses the listed variables (possibly indexed) to denote the elements in various syntactic classes, and use $\{pattern\}$ to signify zero or more repetitions.

Note that the language does not have a special syntax for products and tuples. The programmer may introduce a type of polymorphic pairs, say, and define selectors appropriately. For example,

```
data Pair a b = MkPair a b

fst:: Pair a b -> a
fst z = case z of
           MkPair x y -> x
```

p	∈	*Prog*	[Programs]
TD	∈	*TypeDef*	[Type Definitions]
FT	∈	*FunType*	[Function Types]
FD	∈	*FunDef*	[Function Definitions]
e	∈	*Expr*	[Value Expressions]
x	∈	*Var*	[Value Variables]
f	∈	*Funame*	[Function names]
c	∈	*Cname*	[Constructor names]
t	∈	*Texpr*	[Type Expressions]
a	∈	*Tvar*	[Type variables]
T	∈	*Tname*	[Type names]

```
p    →  {TD} {FT FD}
TD   →  data T {a} = R
FT   →  f::{t ->}t
FD   →  f {x} = e
e    →  x
     |  c {e}
     |  f {e}
     |  case e of
        {c_k {x}-> e}
S    →  {c_1 {t_1}|} |c_n {t_n}
t    →  T {t}
     |  a
R    →  rec a . S | S
```

Fig. 1. Syntactic classes and BNF syntax.

5.2 Semantics

Unlike previous presentations of projection analysis, the semantics is entirely standard. We provide interpretations for types and terms within $\mathscr{C}$.

5.2.1 Denotations of types

We use the standard model of types in lazy functional languages, so the right hand side of a type definition of the form

```
data F a b = C1 R S | C2 T
```

is modelled by the domain

$$(R \times S \; + \; T)_\perp$$

where R, S and T model R, S and T, respectively, the product is cartesian product, and the sum is *disjoint union.* The explicit lifting is required to obtain the behaviour of separated sum. Notice this lifting is nothing to do with the earlier lifting trick to capture strictness, but it does interact with the analysis by turning active demands into latent demands. We model F as a functor from domains to domains, a bi-functor in this case.

As an example, consider lists, defined as follows.

```
data List a = rec L . Nil | Cons a L
```

We model this type by the functor,

$$List = \Lambda\alpha\ .\ \mu L\ .\ (\mathbf{1}\ +\ \alpha \times L)_\perp$$

We write in_{Nil} and in_{Cons} for the lifted injection functions into a named sum (i.e. $in_{Nil} = lift \circ inl$ etc.). Normally, we will drop the explicit $\Lambda\alpha$, and simply use successive Greek letters for successive polymorphic parameters (a sort of de Bruijn index).

5.2.2 Sums and products

We define types using $+$ and $\times$, so we shall provide the definitions of these over partial functions also (and projections in particular).

Suppose $\alpha : A \to A$ and $\beta : B \to B$ are partial functions. Then,

$$\begin{array}{lcll}
(\alpha+\beta)\ (inl\ a) & = & \text{<UNDEF>}, & \textbf{if } \alpha\ a = \text{<UNDEF>} \\
 & = & inl\ (\alpha\ a), & \textbf{otherwise} \\
(\alpha+\beta)\ (inr\ b) & = & \text{<UNDEF>}, & \textbf{if } \beta\ b = \text{<UNDEF>} \\
 & = & inr\ (\beta\ b), & \textbf{otherwise} \\
 & & & \\
(\alpha\times\beta)\ (a,b) & = & \text{<UNDEF>}, & \textbf{if } \alpha\ a = \text{<UNDEF>} \\
 & & & \textbf{or } \beta\ b = \text{<UNDEF>} \\
 & = & (\alpha\ a, \beta\ b), & \textbf{otherwise}
\end{array}$$

The *Both* projection from the introduction can be defined as

$$Both = Abs \sqcup (Str \times Str)$$

so in fact it only recorded latent demand. A strict version, *!Both* which treats a bottom in either component as unacceptable is equal simply to $Str \times Str$.

In the semantics, all uses of $+$ occur only in the presence of an outer lifting (modelling separated sum). From the definition of lifting it is easy to see that $(\alpha+\beta)_\perp$ is total.

5.2.3 Semantics

The semantics are given in terms of two semantic functions,

$$\begin{array}{l}
\mathcal{PR} : FunDef \to FunEnv \\
\mathcal{E} : FunEnv \to Expr \to ValEnv \to Value
\end{array}$$

where

$$\begin{array}{lclcl}
v & \in & Value & = & \bigcup_{\tau\in\text{Type}} D_\tau \\
\rho & \in & ValEnv & = & Var \to Value \\
\phi & \in & FunEnv & = & Funame \to (Value \times \cdots \times Value \to Value)
\end{array}$$

Given a program containing function definitions, the semantic function $\mathcal{PR}$ constructs a global environment of functions. $\mathcal{E}$ interprets expressions in a given function environment. $\mathscr{E}$ is defined as follows.

$$
\begin{array}{lcl}
\mathscr{E}_\phi[\![\, \mathtt{x} \,]\!]_\rho & = & \rho\,(\mathtt{x}) \\[1em]
\mathscr{E}_\phi[\![\, \mathtt{f}\;\; \mathtt{e}_1 \ldots \mathtt{e}_k \,]\!]_\rho & = & \phi\;[\![\, \mathtt{f} \,]\!]\;(\mathscr{E}_\phi[\![\, \mathtt{e}_1 \,]\!]_\rho \times \cdots \times \mathscr{E}_\phi[\![\, \mathtt{e}_k \,]\!]_\rho) \\[1em]
\mathscr{E}_\phi[\![\, \mathtt{c}\;\; \mathtt{e}_1 \ldots \mathtt{e}_k \,]\!]_\rho & = & \mathit{in}_{\mathtt{c}}\;(\mathscr{E}_\phi[\![\, \mathtt{e}_1 \,]\!]_\rho \times \cdots \times \mathscr{E}_\phi[\![\, \mathtt{e}_k \,]\!]_\rho)
\end{array}
$$

$$
\begin{array}{l}
\mathscr{E}_\phi[\![\, \mathtt{case}\;\; \mathtt{e}\;\; \mathtt{of}\;\; \cdots \mathtt{c}_j\; \mathtt{x}_1 \ldots \mathtt{x}_k\; \texttt{->}\; \mathtt{e}_j \cdots \,]\!]_\rho \\
\quad = \quad \mathit{case}\; \mathscr{E}_\phi[\![\, \mathtt{e} \,]\!]_\rho\; \mathit{of} \\
\qquad\qquad\quad \begin{array}{lcl}
\bot & \to & \bot \\
\vdots & & \\
\mathit{in}_{\mathtt{c}_j}\;(v_1 \times \cdots \times v_k) & \to & \mathscr{E}_\phi[\![\, \mathtt{e}_j \,]\!]_{\rho[\mathtt{x}_i \mapsto v_i]} \\
\vdots & &
\end{array}
\end{array}
$$

The function environment is constructed as the least fixed point of the function definitions, as follows,

$$
\mathscr{P}\mathscr{R}[\![\, \cdots,\; \mathtt{f}\;\; \mathtt{x}_1\;\; ..\;\; \mathtt{x}_k\; \texttt{=}\; \mathtt{e},\; \cdots \,]\!] = \mathit{fix}\;(\lambda\phi.\{\cdots, f \mapsto \lambda(v_1, \ldots, v_k).\mathscr{E}_\phi[\![\, \mathtt{e} \,]\!]_{[\mathtt{x}_i \mapsto v_i]}, \cdots\})
$$

5.3 Analysis rules

In this section we define an abstract semantics for our language to perform backward strictness analysis. It takes the form of a projection transformer, and corresponds directly with the analysis of Kubiak *et al.* (1992) except, of course, the absence of explicit liftings.

We extend the operations of $\sqcup$ and & to denote corresponding point-wise operations on abstract environments. In such an environment

$$
\mathit{env}\;\in\;\mathit{AbsEnv} = \mathit{Var} \to \mathit{Proj}
$$

names are associated with partial projections. We use [ ] to denote the initial environment in which every identifier is mapped to *Fail*. By $[(\mathtt{x}, \alpha)]$ we mean the initial environment extended by binding the variable x to the projection α, and by $\rho \setminus \{\mathtt{x}_1, \ldots, \mathtt{x}_k\}$ we denote the environment differing from ρ in that the variables $\{\mathtt{x}_1, \ldots, \mathtt{x}_k\}$ are mapped to *Fail*. For any environments ρ_1 and ρ_2, if x does not belong to the domain of ρ_1 then $(\rho_1 \sqcup \rho_2)(\mathtt{x}) = \rho_2(\mathtt{x})$. We also treat & in a similar way.

The *projection transformer* $\mathscr{E}^{\#}$, takes an expression e and a projection α (which expresses the demand on the value of e), and builds an environment $\rho = \mathscr{E}^{\#}_{\phi^{\#}}[\![\, \mathtt{e} \,]\!]\alpha$ in which all free variables $\mathtt{x}_i$ of e are assigned projections. The environment ρ is constructed so that the following *safety* condition is satisfied for all projections α of appropriate type:

$$
\begin{array}{c}
\alpha \;\circ\; \lambda(v_1 \times \cdots \times v_k).\mathscr{E}_\phi[\![\, \mathtt{e} \,]\!]_{[\mathtt{x}_i \mapsto v_i]} \\
\sqsubseteq \\
\lambda(v_1 \times \cdots \times v_k).\mathscr{E}_\phi[\![\, \mathtt{e} \,]\!]_{[\mathtt{x}_i \mapsto v_i]} \;\circ\; (\rho(\mathtt{x}_1), \ldots, \rho(\mathtt{x}_k))
\end{array}
$$

The projection $\rho(\mathrm{x}_i)$ is the demand on parameter x_i given a demand α on the value of e, so this condition is just a multi-argument generalisation of the condition given in section 2. The best possible environment is one in which variables are associated with the least projections for which the safety condition is still guaranteed.

Note that the composition operator is composition of partial functions (i.e. in $\mathscr{P}$) and that the total function $\lambda(v_1,\ldots,v_k).\mathscr{E}_\phi[\![\mathrm{e}]\!]_{[\mathrm{x}_i\mapsto v_i]}$ is embedded into $\mathscr{P}$. In particular, if we apply the function to a product element $(a_1,\ldots,a_k)$ and it turns out that for some i, $\rho(\mathrm{x}_i)a_i$ is undefined, then

$$(\rho(\mathrm{x}_1)\times\cdots\times\rho(\mathrm{x}_k))\ (a_1,\ldots,a_k) = <\text{UNDEF}>$$

also, and so by definition of composition in $\mathscr{P}$, the whole expression is undefined.

Our first use of partiality is in the definition of the projection transformer $\mathscr{E}^\#$. The first equation realises the guard operation from Wadler and Hughes (1987).

$$\mathscr{E}^\#_{\phi^\#}[\![\mathrm{e}]\!]\ \alpha \quad = \quad \mathscr{E}^\#_{\phi^\#}[\![\mathrm{e}]\!]\,!\alpha \ \sqcup\ \lambda x.Abs$$

Note that all partial projections may be expressed in the form either α or $!\alpha$ (where α is total). Recall, the intuition behind a latent demand α (a total projection) is, "this value may or may not be required, but if it is then α's worth will be needed." Conversely, a demand of the form $!\alpha$ means, "this value *will* be required, and what's more, α's worth of it will be needed". With this intuition, the equation above may be read as follows, "to compute the demand propagated from a lazy demand, first compute it as if the demand was strict, and then make all the resulting demands lazy".

The rest of the equations apply to projections expressible as $!\alpha$.

$$\mathscr{E}^\#_{\phi^\#}[\![\mathrm{x}]\!]\,!\alpha \quad = \quad [(\mathrm{x}, !\alpha)]$$

$$\begin{aligned}
&\mathscr{E}^\#_{\phi^\#}[\![\mathrm{f\ e_1\ldots e}_k]\!]\,!\alpha \\
&\qquad = \quad \mathscr{E}^\#_{\phi^\#}[\![\mathrm{e}_1]\!]\alpha_1\ \&\cdots\&\ \mathscr{E}^\#_{\phi^\#}[\![\mathrm{e}_k]\!]\alpha_k \\
&\qquad\qquad \textbf{where } [(\mathrm{x}_1,\alpha_1),\ldots,(\mathrm{x}_k,\alpha_k)] = (\phi^\#[\![\mathrm{f}]\!])\ !\alpha \\
&\qquad\qquad \textbf{and } \mathrm{x}_1,\ldots,\mathrm{x}_k \textbf{ are the parameters of f}
\end{aligned}$$

$$\begin{aligned}
&\mathscr{E}^\#_{\phi^\#}[\![\mathrm{c\ e_1\ldots e}_k]\!]\,!(\cdots+\ c:(\alpha_1\times\cdots\times\alpha_k)+\cdots) \\
&\qquad = \quad \mathscr{E}^\#_{\phi^\#}[\![\mathrm{e}_1]\!]\alpha_1\ \&\cdots\&\ \mathscr{E}^\#_{\phi^\#}[\![\mathrm{e}_k]\!]\alpha_k
\end{aligned}$$

$$\begin{aligned}
&\mathscr{E}^\#_{\phi^\#}[\![\mathrm{case\ e\ of}\ \cdots \mathrm{c}_i\ \mathrm{x}_1\ldots\ \mathrm{x}_k\text{->}\mathrm{e}_i\cdots\ ]\!]\,!\alpha \\
&\qquad = \quad \textstyle\bigsqcup_i\ (\mathscr{E}^\#_{\phi^\#}[\![\mathrm{e}]\!]\,!(\delta_i)\ \&\ \rho_i \setminus \{\mathrm{x}_1,\ldots,\mathrm{x}_k\}) \\
&\textbf{where} \\
&\quad \rho_i = \mathscr{E}^\#_{\phi^\#}[\![\mathrm{e}_i]\!]\,!\alpha \\
&\quad \delta_i = \mathit{Fail}+\cdots+(\rho_i(\mathrm{x}_1)\times\cdots\times\rho_i(\mathrm{x}_k))+\cdots+\mathit{Fail}
\end{aligned}$$

In these rules the structure of the contexts corresponds to their underlying types. For example, in the rule for constructor applications the projection is a sum of products of projections; the particular summand given is associated with the constructor c.

Similarly, the *Fail*s appearing in the rule for case-expressions should be understood as the bottom projections over the target types of the remaining constructors, different from c.

We now go back to the proof of the safety condition.

Proposition
Suppose ϕ and $\phi^\#$ are function environments such that

$$\alpha \circ \phi[\![\mathtt{f}]\!] \sqsubseteq \phi[\![\mathtt{f}]\!] \circ ((\phi^\#[\![\mathtt{f}]\!]\alpha)(\mathtt{x}_1) \times \ldots \times (\phi^\#[\![\mathtt{f}]\!]\alpha)(\mathtt{x}_n))$$

for all function names f and all partial projections α (of the appropriate type), where $\mathtt{x}_1, \ldots, \mathtt{x}_n$ are the parameters of f. Then for any expression e

$$\begin{array}{c} \alpha \quad \circ \quad \lambda(v_1, \ldots, v_k).\mathscr{E}_\phi[\![\mathtt{e}]\!]_{[\mathtt{y}_i \mapsto v_i]} \\ \sqsubseteq \\ \lambda(v_1, \ldots, v_k).\mathscr{E}_\phi[\![\mathtt{e}]\!]_{[\mathtt{y}_i \mapsto v_i]} \quad \circ \quad (\rho(\mathtt{y}_1) \times \cdots \times \rho(\mathtt{y}_k)) \end{array}$$

where $\rho = \mathscr{E}^\#_{\phi\#}[\![\mathtt{e}]\!]\alpha$.

Proof
The proof is given by structural induction on e. It is sufficient to show that

$$\alpha(\mathscr{E}_\phi[\![\mathtt{e}]\!]_{[\mathtt{y}_i \mapsto v_i]}) \sqsubseteq \mathscr{E}_\phi[\![\mathtt{e}]\!]_{[\mathtt{y}_i \mapsto \rho(\mathtt{y}_i)v_i]}$$

From our earlier comments, recall that if $\mathtt{y}_i \mapsto \rho(\mathtt{y}_i)v_i = \langle\text{UNDEF}\rangle$ for any i then the expression $\mathscr{E}_\phi[\![\mathtt{e}]\!]_{[\mathtt{y}_i \mapsto \rho(\mathtt{y}_i)v_i]} = \langle\text{UNDEF}\rangle$.

`Case: (y`$_i$`)`

It is easy to observe that in this case we actually have equality.

`Case: (f e`$_1$ `... e`$_k$`)`

Let $\rho = \mathscr{E}^\#_{\phi\#}[\![\mathtt{f}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_k]\!]\alpha$, $\sigma = \phi^\#[\![\mathtt{f}]\!]\alpha$ and $\tau_i = \mathscr{E}^\#_{\phi\#}[\![\mathtt{e}_i]\!](\sigma(\mathtt{x}_i))$. Now,

$$\begin{array}{ll} & \alpha(\mathscr{E}_\phi[\![\mathtt{f}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_k]\!]_{[\mathtt{y}_i \mapsto v_i]}) \\ = & \alpha(\phi[\![\mathtt{f}]\!](\mathscr{E}_\phi[\![\mathtt{e}_1]\!]_{[\mathtt{y}_i \mapsto v_i]} \times \ldots \times \mathscr{E}_\phi[\![\mathtt{e}_k]\!]_{[\mathtt{y}_i \mapsto v_i]})) \\ \sqsubseteq & \phi[\![\mathtt{f}]\!](\sigma(\mathtt{x}_1)(\mathscr{E}_\phi[\![\mathtt{e}_1]\!]_{[\mathtt{y}_i \mapsto v_i]}) \times \ldots \times \sigma(\mathtt{x}_k)(\mathscr{E}_\phi[\![\mathtt{e}_k]\!]_{[\mathtt{y}_i \mapsto v_i]})) \\ \sqsubseteq & \phi[\![\mathtt{f}]\!](\mathscr{E}_\phi[\![\mathtt{e}_1]\!]_{[\mathtt{y}_i \mapsto \tau_1(\mathtt{y}_i)v_i]} \times \ldots \times \mathscr{E}_\phi[\![\mathtt{e}_k]\!]_{[\mathtt{y}_i \mapsto \tau_k(v_i)v_i]}) \end{array}$$

If for any i and j, $\tau_j(\mathtt{y}_i)v_i$ is $\langle\text{UNDEF}\rangle$, then the last expression in the inequality above is also $\langle\text{UNDEF}\rangle$, and hence the result holds. Otherwise, from the definition of ρ we have $\tau_j(\mathtt{y}_i)v_i \sqsubseteq \rho(\mathtt{y}_i)v_i$. Moreover,

$$\mathscr{E}_\phi[\![\mathtt{f}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_k]\!]_{[\mathtt{y}_i \mapsto \rho(\mathtt{y}_i)v_i]} = \phi[\![\mathtt{f}]\!](\mathscr{E}_\phi[\![\mathtt{e}_1]\!]_{[\mathtt{y}_i \mapsto \rho(\mathtt{y}_i)v_i]} \times \ldots \times \mathscr{E}_\phi[\![\mathtt{e}_k]\!]_{[\mathtt{y}_i \mapsto \rho(\mathtt{y}_i)v_i]})$$

Hence we have the desired result.

`Case: (c e`$_1$ `... e`$_k$`)`

We may assume that $\alpha = !(\cdots + c : (\alpha_1 \times \cdots \times \alpha_k) + \cdots)$. Now,

$$\begin{array}{ll} & \alpha(\mathscr{E}_\phi[\![\mathtt{c}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_k]\!]_{[\mathtt{y}_i \mapsto v_i]}) \\ = & \alpha(in_{\mathtt{c}}(\mathscr{E}_\phi[\![\mathtt{e}_1]\!]_{[\mathtt{y}_i \mapsto v_i]} \times \ldots \times \mathscr{E}_\phi[\![\mathtt{e}_k]\!]_{[\mathtt{y}_i \mapsto v_i]})) \\ = & in_{\mathtt{c}}(\alpha_1(\mathscr{E}_\phi[\![\mathtt{e}_1]\!]_{[\mathtt{y}_i \mapsto v_i]}) \times \ldots \times \alpha_k(\mathscr{E}_\phi[\![\mathtt{e}_k]\!]_{[\mathtt{y}_i \mapsto v_i]})) \\ \sqsubseteq & in_{\mathtt{c}}(\mathscr{E}_\phi[\![\mathtt{e}_1]\!]_{[\mathtt{y}_i \mapsto \tau_1(\mathtt{y}_i)v_i]} \times \ldots \times \mathscr{E}_\phi[\![\mathtt{e}_k]\!]_{[\mathtt{y}_i \mapsto \tau_k(\mathtt{y}_i)v_i]}) \end{array}$$

where $\tau_j = \mathscr{E}^{\#}_{\phi\#}[\![\mathtt{e}_j]\!]\alpha_j$. On the other hand, if

$$\begin{aligned} \rho &= \mathscr{E}^{\#}_{\phi\#}[\![\mathtt{c\ e}_1\ \ldots\ \mathtt{e}_k]\!]\alpha \\ &= \mathscr{E}^{\#}_{\phi\#}[\![\mathtt{e}_1]\!]\alpha_1\ \&\cdots\&\ \mathscr{E}^{\#}_{\phi\#}[\![\mathtt{e}_k]\!]\alpha_k \end{aligned}$$

then,

$$\begin{aligned} &\mathscr{E}_\phi[\![\mathtt{c\ e}_1\ \ldots\ \mathtt{e}_k]\!]_{[\mathtt{y}_i \mapsto \rho(\mathtt{y}_i)v_i]} \\ &\quad = inc_{\mathtt{c}}(\mathscr{E}_\phi[\![\mathtt{e}_1]\!]_{[\mathtt{y}_i \mapsto \rho(\mathtt{y}_i)v_i]} \times \ldots \times \mathscr{E}_\phi[\![\mathtt{e}_k]\!]_{[\mathtt{y}_i \mapsto \rho(\mathtt{y}_i)v_i]}) \end{aligned}$$

As in the previous case, if $\tau_j(\mathtt{y}_i)v_i$ is <UNDEF> for some i and j then so is the value on the right hand side of the inequality above. We may therefore assume that none of the values is undefined, so that $\tau_j(\mathtt{y}_i)v_i \sqsubseteq \rho(\mathtt{y}_i)v_i$. From this, we obtain the required inequality.

`Case:` $(\mathtt{case\ e\ of}\ \cdots\ \mathtt{c}_j\ \mathtt{x}_1 \ldots \mathtt{x}_k\ \mathtt{->}\ \mathtt{e}_j \cdots)$

There are two possibilities. The first is where $\mathscr{E}_\phi[\![\mathtt{e}]\!]_{[\mathtt{y}_i \mapsto v_i]} = \bot$. In this case, the left hand side of the inequality we want to prove becomes <UNDEF>, and hence we have the desired result. The other is the case where we may have

$$\mathscr{E}_\phi[\![\mathtt{e}]\!]_{[\mathtt{y}_i \mapsto v_i]} = inc_{\mathtt{c}_j}\ u_1 \times \ldots \times u_k$$

Now,

$$\begin{aligned} &\alpha(\mathscr{E}_\phi[\![\mathtt{case\ e\ of}\ \cdots\ \mathtt{c}_j\ \mathtt{x}_1 \ldots \mathtt{x}_k\ \mathtt{->}\ \mathtt{e}_j \ldots]\!]_{[\mathtt{y}_i \mapsto v_i]}) \\ &\quad = \alpha(\mathscr{E}_\phi[\![\mathtt{e}_j]\!]_{[\mathtt{y}_i \mapsto v_i, \mathtt{x}_s \mapsto u_s]}) \\ &\quad \sqsubseteq \mathscr{E}_\phi[\![\mathtt{e}_j]\!]_{[\mathtt{y}_i \mapsto \rho_j(\mathtt{y}_i)v_i, \mathtt{x}_s \mapsto \rho_j(\mathtt{x}_s)u_s]} \end{aligned}$$

where $\rho_j = \mathscr{E}^{\#}_{\phi\#}[\![\mathtt{e}_j]\!]\alpha$. Let

$$\begin{aligned} \rho &= \textstyle\bigsqcup_j\ \sigma_j\ \&\ \tau_j \\ \tau_j &= \rho_j \setminus \{\mathtt{x}_1, \ldots, \mathtt{x}_k\} \\ \sigma_j &= \mathscr{E}^{\#}_{\phi\#}[\![\mathtt{e}]\!]\ !(\delta_j) \\ \delta_j &= \mathit{Fail} + \cdots + (\rho_j(\mathtt{x}_1) \times \cdots \times \rho_j(\mathtt{x}_k)) + \cdots + \mathit{Fail} \end{aligned}$$

We now have

$$\begin{aligned} inc_{\mathtt{c}_j}\ \rho_j(\mathtt{x}_1)u_1 \times \ldots \times \rho_j(\mathtt{x}_k)u_k &= !(\delta_j)\ (inc_{\mathtt{c}_j} u_1 \times \ldots \times u_k) \\ &= !(\delta_j)\ (\mathscr{E}_\phi[\![\mathtt{e}]\!]_{[\mathtt{y}_i \mapsto v_i]}) \\ &\sqsubseteq \mathscr{E}_\phi[\![\mathtt{e}]\!]_{[\mathtt{y}_i \mapsto \sigma_j(\mathtt{y}_i)v_i]} \\ &\sqsubseteq \mathscr{E}_\phi[\![\mathtt{e}]\!]_{[\mathtt{y}_i \mapsto \rho(\mathtt{y}_i)v_i]} \\ &= inc_{\mathtt{c}_j} w_1 \times \ldots \times w_k \end{aligned}$$

for some $w_1\ \ldots\ w_k$. Notice that

$$\begin{aligned} &\mathscr{E}_\phi[\![\mathtt{case\ e\ of}\ \cdots\ \mathtt{c}_j\ \mathtt{x}_1 \ldots \mathtt{x}_k\ \mathtt{->}\ \mathtt{e}_j \ldots]\!]_{[\mathtt{y}_i \mapsto \rho(\mathtt{y}_i)v_i]} \\ &\quad = \mathscr{E}_\phi[\![\mathtt{e}_j]\!]_{[\mathtt{y}_i \mapsto \rho(\mathtt{y}_i)v_i, \mathtt{x}_s \mapsto w_s]} \\ &\quad \sqsupseteq \mathscr{E}_\phi[\![\mathtt{e}_j]\!]_{[\mathtt{y}_i \mapsto \rho_j(\mathtt{y}_i)v_i, \mathtt{x}_s \mapsto \rho_j(\mathtt{x}_s)u_s]} \end{aligned}$$

Therefore, we have the desired result. □

Because of the remark we made about $\mathscr{E}^{\#}$ we only need to construct the function environment for active demands. Again it is constructed as the least fixed point of the function definitions.

$$\mathscr{PR}^{\#}[\![\cdots,\ \mathtt{f\ x}_1\ ..\ \mathtt{x}_k\ =\ \mathtt{e},\ \cdots]\!] = \mathit{fix}\ (\lambda\phi^{\#}.\{\cdots, f \mapsto \lambda\alpha\ .\ \mathscr{E}^{\#}_{\phi^{\#}}[\![\mathtt{e}]\!]\ \alpha, \cdots\})$$

It is necessary to prove that the function environments obtained satisfy the hypothesis of the preceding proposition. For simplicity, we assume that we only have the single function definition `f x = e`.

Proposition

Suppose ϕ and $\phi^{\#}$ are function environments which respectively arise from the standard semantics and the abstract semantics. Let α be a partial projection of the appropriate type. Then

$$\alpha \circ \phi[\![\mathtt{f}]\!] \sqsubseteq \phi[\![\mathtt{f}]\!] \circ ((\phi^{\#}[\![\mathtt{f}]\!]\alpha)(\mathtt{x}))$$

Proof

The function environments ϕ and $\phi^{\#}$ are the respective limits of the sequences $\{\phi_n\}$ and $\{\phi^{\#}_n\}$ where

$$\begin{array}{lcl}\phi_0[\![\mathtt{f}]\!] & = & \lambda v\ .\ \bot \\ \phi_{n+1}[\![\mathtt{f}]\!] & = & \lambda v\ .\ \mathscr{E}_{\phi_n}[\![\mathtt{e}]\!]_{[\mathtt{x}\mapsto v]}\end{array}$$

$$\begin{array}{lcl}(\phi^{\#}_0[\![\mathtt{f}]\!]\alpha)(\mathtt{x}) & = & \mathit{Fail} \\ (\phi^{\#}_{n+1}[\![\mathtt{f}]\!]\alpha)(\mathtt{x}) & = & (\mathscr{E}^{\#}_{\phi^{\#}_n}[\![\mathtt{e}]\!]\alpha)(\mathtt{x})\end{array}$$

Let $\delta_n = (\phi^{\#}_n[\![\mathtt{f}]\!]\alpha)(\mathtt{x})$ and $\delta = (\phi^{\#}[\![\mathtt{f}]\!]\alpha)(\mathtt{x})$. First, we will show that

$$\alpha \circ \phi_n[\![\mathtt{f}]\!] \sqsubseteq \phi_n[\![\mathtt{f}]\!] \circ ((\phi^{\#}[\![\mathtt{f}]\!]\alpha)(\mathtt{x}))$$

by induction on n.

`Case:` *Base*

Since α is a partial projection, the left hand side of the inequality is <UNDEF> when $n = 0$.

`Case:` *Inductive*

$$\begin{array}{rcl}\phi_{n+1}[\![\mathtt{f}]\!] \circ \delta & \sqsupseteq & \phi_{n+1}[\![\mathtt{f}]\!] \circ \delta_{n+1} \\ & = & \phi_{n+1}[\![\mathtt{f}]\!] \circ (\mathscr{E}^{\#}_{\phi^{\#}_n}[\![\mathtt{e}]\!]\alpha)(\mathtt{x}) \\ & = & (\lambda v\ .\ \mathscr{E}_{\phi_n}[\![\mathtt{e}]\!]_{[\mathtt{x}\mapsto v]}) \circ (\mathscr{E}^{\#}_{\phi^{\#}_n}[\![\mathtt{e}]\!]\alpha)(\mathtt{x}) \\ & = & \lambda v\ .\ \mathscr{E}_{\phi_n}[\![\mathtt{e}]\!]_{[\mathtt{x}\mapsto(\mathscr{E}^{\#}_{\phi^{\#}_n}[\![\mathtt{e}]\!]\alpha)(\mathtt{x})v]} \\ & \sqsupseteq & \lambda v\ .\ \alpha(\mathscr{E}_{\phi_n}[\![\mathtt{e}]\!]_{[\mathtt{x}\mapsto v]}) \\ & = & \alpha \circ (\lambda v\ .\ \mathscr{E}_{\phi_n}[\![\mathtt{e}]\!]_{[\mathtt{x}\mapsto v]}) \\ & = & \alpha \circ \phi_{n+1}[\![\mathtt{f}]\!]\end{array}$$

`Case:` *Limit*

$$\begin{aligned}\phi[\![\mathtt{f}]\!]\circ\delta &= (\textstyle\bigsqcup_n\ \phi_n[\![\mathtt{f}]\!])\circ\delta\\ &= \textstyle\bigsqcup_n\ (\phi_n[\![\mathtt{f}]\!]\circ\delta)\\ &\sqsupseteq \textstyle\bigsqcup_n\ \alpha\circ\phi_n[\![\mathtt{f}]\!]\\ &= \alpha\circ(\textstyle\bigsqcup_n\ (\phi_n[\![\mathtt{f}]\!])\\ &= \alpha\circ\phi[\![\mathtt{f}]\!]\end{aligned}$$

This completes the proof. □

5.4 *Results of analysis*

The familiar head-strict and tail-strict projections over the list type $\mu L\ .\ (\mathbf{1}\ +\ \alpha\times L)_\perp$ are given by

$$\begin{aligned}Head &= \mu l\ .\ (\mathbf{1}\ +\ !\alpha\times l)_\perp\\ Tail &= \mu l\ .\ (\mathbf{1}\ +\ \alpha\times !l)_\perp\end{aligned}$$

Strictly speaking, the polymorphic projections are represented by (*Head Ide*) and (*Tail Ide*) using the de-Briujn convention mentioned earlier, but we will often be sloppy and understand that uninstantiated parameters α, β etc. are actually instantiated to *Ide*.

Below we present results obtained by the strictness analysis defined above. In the projections which follow, we write the constructor names in explicitly to aid understanding.

5.4.1 *Lists*

We begin with some standard list-based examples.

```
data List a = rec L . Nil | Cons a L

append :: List a -> List a -> List a
append xs zs
  = case xs of
      Nil -> zs
      Cons y ys -> Cons y (append ys zs)

reverse :: List a -> List a
reverse rs
  = case rs of
      Nil -> Nil
      Cons y ys -> append (reverse ys)
                          (Cons y Nil)
```

First *append*. Consider the demand

$$!(\mu l\ .\ (Nil:\mathbf{1}\ +\ Cons:!\alpha\times l)_\perp)$$

for *append*'s result. This is an active demand (hence the outer application of partiality) which is recursively active each list element (hence the partiality on the α), but latent in the list tails (the non-partiality of l). This is what we previously wrote as $!Head$. From this result context, the demand on *append*'s arguments is computed as,

$$[\texttt{xs} \mapsto !(\mu l \,.\, (Nil : \mathbf{1} \;+\; Cons : !\alpha \times l)_\perp), \texttt{zs} \mapsto \mu l \,.\, (Nil : \mathbf{1} \;+\; Cons : !\alpha \times l)_\perp]$$

In summary, a strict, head-strict demand $!Head$ for *append*'s result is translated to a strict and head-strict demand for the first argument, and a lazy, head-strict demand for the second, i.e. $!Head \times Head$.

Alternatively, given a demand

$$!(\mu l \,.\, (Nil : \mathbf{1} \;+\; Cons : \alpha \times !l)_\perp)$$

(that is, strict and tail-strict, $!Tail$) for the result of *append*, the analysis deduces a demand of

$$[\texttt{xs} \mapsto !(\mu l \,.\, (Nil : \mathbf{1} \;+\; Cons : \alpha \times !l)_\perp), \texttt{zs} \mapsto !(\mu l \,.\, (Nil : \mathbf{1} \;+\; Cons : \alpha \times !l)_\perp)]$$

for its arguments, i.e. both arguments strict and tail-strict, $!Tail \times !Tail$.

The analyser obtained the following facts about *reverse*. If its result is demanded in a strict and head-strict context

$$!(\mu l \,.\, (Nil : \mathbf{1} \;+\; Cons : !\alpha \times l)_\perp)$$

then its argument is in a strict and tail-strict context

$$[\texttt{rs} \mapsto !(\mu l \,.\, (Nil : \mathbf{1} \;+\; Cons : \alpha \times !l)_\perp)]$$

Likewise, if the result is demanded strictly and tail-strictly, then so is its argument. Combining these facts, we see that *reverse* is strict and tail-strict, in both $!Head$ and $!Tail$ contexts.

5.4.2 *Trees*

For the next examples we introduce a polymorphic tree type. As with lists, the contexts over trees are generated automatically, having a structure which corresponds to the structure of the type definition.

```
data Tree a = rec T . Leaf a | Branch T T

flatten :: Tree a -> List a
flatten t = case t of
              Leaf x     -> Cons x Nil
              Branch l r -> append (flatten l) (flat r)
```

Whereas particular contexts over lists like *Head* and *Tail* have standard names, allowing the results of the analysis to be written compactly, contexts over trees do not. However, as with the list contexts, it is very easy to read the strictness from the contexts.

The function *flatten* collapses a tree down to a list. If that result list is demanded by a strict, and head-strict context, $!Head$, that is,

$$!(\mu l\ .\ (Nil\ :\mathbf{1}\ +\ Cons\ :!\alpha \times l)_\perp)$$

then the analyser deduces a demand on the tree argument of,

$$[\mathtt{t} \mapsto\ !(\mu t\ .\ (Leaf\ :!\alpha\ +\ Branch\ :!t \times t)_\perp)]$$

That is, a strict, *Leaf*-strict, left-strict context. When a tree is built in such a context its left spine may be constructed strictly, all the way down to the leaf. The rest of the tree is left unevaluated. If any other part of the tree is required, again its left spine is evaluated all the way to the leaf, and so on.

Alternatively, if the result of *flatten* is demanded in a strict, and tail-strict context, $!Tail$, that is,

$$!(\mu l\ .\ (Nil\ :\mathbf{1}\ +\ Cons\ :\alpha \times !l)_\perp)$$

then the demand on *flat*'s argument is,

$$[\mathtt{xs} \mapsto\ !(\mu t\ .\ (Leaf\ :\alpha\ +\ Branch\ :!t \times !t)_\perp)]$$

which is a strict and left-and-right-strict context. The structure of the tree will be evaluated, but none of the leaves.

5.4.3 *Instances of polymorphic functions*

All the examples so far have been of polymorphic functions on their own. The final series of examples are of instances of polymorphic functions. We define a type of Peano numerals together with addition, and use these to define a function which sums the leaves of a tree.

```
data Nat = rec N . Zero | Succ N

add :: Nat -> Nat -> Nat
add a b = case a of
            Zero -> b
            Succ c -> Succ (add c b)

sum :: Tree Nat -> Nat
sum t = case t of
          Leaf x   -> x
          Node l r -> add (sum l) (sum r)
```

Because the numerals are non-atomic, *add* has some interesting strictness. A result context of

$$!(\mu n\ .\ (Zero\ :\mathbf{1}\ +\ Succ\ :!n)_\perp)$$

produces an argument context of

$$[\mathtt{a} \mapsto\ !(\mu n\ .\ (\mathit{Zero} : \mathbf{1}\ +\ \mathit{Succ} : !n)_\perp), \mathtt{b} \mapsto\ !(\mu n\ .\ (\mathit{Zero} : \mathbf{1}\ +\ \mathit{Succ} : !n)_\perp)]$$

Conversely, a result context of

$$!(\mu n\ .\ (\mathit{Zero} : \mathbf{1}\ +\ \mathit{Succ} : n)_\perp)$$

generates the argument context

$$[\mathtt{a} \mapsto\ !(\mu n\ .\ (\mathit{Zero} : \mathbf{1}\ +\ \mathit{Succ} : n)_\perp), \mathtt{b} \mapsto \mu n\ .\ (\mathit{Zero} : \mathbf{1}\ +\ \mathit{Succ} : n)_\perp]$$

In other words, if the complete result of *add* is demanded then both its arguments are demanded completely. Conversely, if the result of *add* is only demanded strictly, then its first argument is demanded strictly, but it's second lazily.

Now let us examine *sum*. If *sum*'s result is demanded hyper-strictly then the analyser deduces that its argument is also demanded hyper-strictly. That is, a result context,

$$!(\mu n\ .\ (\mathit{Zero} : \mathbf{1}\ +\ \mathit{Succ} : !n)_\perp)$$

is converted to the argument context,

$$[\mathtt{t} \mapsto ((\mu t\ .\ (\mathit{Leaf} : !\alpha\ +\ \mathit{Node} : !t \times !t)_\perp)\ !(\mu n\ .\ (\mathit{Zero} : \mathbf{1}\ +\ \mathit{Succ} : !n)_\perp))]$$

Notice the explicit application of one context to another. As we mentioned earlier, all the polymorphic contexts like H and T should actually have been applied to *Ide*, but this would have cluttered up the examples unnecessarily. In this case, the argument projection should be substituted for α.

As a final example, suppose the demand for *sum*'s result is merely strict. Then the demand for *sum*'s tree argument is strict, leaf-strict and left-strict. That is, the analyser converts the result context

$$!(\mu n\ .\ (\mathit{Zero} : \mathbf{1}\ +\ \mathit{Succ} : n)_\perp)$$

to the argument context

$$[\mathtt{t} \mapsto ((\mu t\ .\ (\mathit{Leaf} : !\alpha\ +\ \mathit{Node} : !t \times t)_\perp)\ !(\mu n\ .\ (\mathit{Zero} : \mathbf{1}\ +\ \mathit{Succ} : n)_\perp))]$$

6 Relationship to other work

6.1 Unboxed values

Peyton Jones and Launchbury (1991) showed that *unboxed values* are useful source-level additions to a lazy functional language, certainly as far as compiler optimisations are concerned. They allow both flow of control and representation issues to be exposed to the compiler while remaining purely within the functional framework.

For example, consider the `double` function defined as follows.

```
double x = x+x
```

In applications of `double`, unless something clever is done, most implementations would attempt to evaluate the argument twice, even though on the second attempt it is certain to be in normal form. If, however, integers and the + function are defined in terms of yet more primitive operations, then standard program transformation can remove this double evaluation. We define,

```
data Int = MkInt Int#

m + n = case m of MkInt m# ->
          case n of MkInt n# ->
            case m# +# n# of r# -> MkInt r#
```

where Int# is the type of unboxed integers. The # on variable names is for human-readability only, and is used to indicate an unboxed value. Now by unfolding the definition of +, the definition of double becomes:

```
double x = case x of MkInt m# ->
             case x of MkInt n# ->
               case m# +# n# of r# -> MkInt r#
```

which, by eliminating a repeated case, becomes,

```
double x = case x of MkInt m# ->
             case m# +# m# of r# -> MkInt r#
```

More generally, if a (recursive) function `f::Int->Int`, say, is known to be strict then it is factorised into a worker and a wrapper:

```
f x = case x of MkInt n# -> f# n#

f# n# = ....
```

where the body of `f#` contains the work done in `f` originally. Typically, the wrapper is unfolded and disappears, leading to very efficient code which has taken advantage of strictness.

To give a semantics for unboxed types, *unpointed domains* were introduced. As we have seen, these are domains which lack a bottom element. For example, the type of unboxed integers, called `int#`, is precisely the set of integers (actually some middle portion of the set), i.e. no bottom element.

The tie-up with the work here is that partial projections lead automatically to the *same domains*. The image of *Str* when applied to *Int*, for example, is exactly `Int#`. This is very exciting, because it means that the results of strictness analysis should now be able to be tied very closely to existing implementation technology. However, it is still early days, and the implications need to be worked out in detail.

6.2 PERs

Every function specifies an equivalence relation: two values are considered equivalent if the function maps them to the same point. Projections are functions so the same thing applies. Moreover, in this case, it works the other way round as well: the equivalence relation specifies the projection completely.

This insight suggests that analyses based on equivalence relations ought to subsume projection-based analyses, and indeed it does. In his thesis, Hunt (1991) defined strictness analysis in terms of *partial* equivalence relations – a generalisation forced by moving to the higher-order case. Partial equivalence relations have no reflexivity axiom, so a point may be unrelated to anything, including itself.

At ground types, Hunt chose exactly those PERs which arise from the projections used in projection analysis, with the addition of one other (which he called BOT) that was used to capture simple strictness – unsurprisingly (given the projection background), the other relations could not capture simple strictness. The PERs at higher types were all induced from these using logical relations between PERs of lower types.

This structure is very elegant with one glaring exception. BOT is quite anomalous compared with the rest of the scheme. In the PER ordering it is less than ID (which stands for 'no information') whereas all other strictness PERs are greater than ID. Because of this, it does not combine with other strictness patterns. For example, head-strictness can be expressed by a single PERs, but 'strict and head-strict' cannot (unlike in the projection world).

An improvement suggested by this paper is to take PERs corresponding to *partial projections*. Then the PER corresponding with STR will detect strictness. It is in the right place in the domain, and can be combined with other strictness patterns.

There is one problem with this. In order to use the obvious model for *fix*, Hunt placed a restriction on his PERs, namely that they should all be strict (relate $\perp$ to $\perp$) and inductive (if the corresponding elements of two chains are related, then so are the limits). Unfortunately, the PER corresponding with STR is not strict as it does not relate $\perp$ to anything.

What is the solution? The problem seems to come from the model for the type of *fix*, namely $\forall A\ .\ (A \rightarrow A) \rightarrow A$. If function spaces are not lifted, then there seems no choice but to use Hunt's scheme with strict PERs. On the other hand, if function spaces are lifted as advocated by Abramsky (1990) and Ong (1988), the problem seems to vanish. Interestingly, Davis (1994) found it essential to use lifted function spaces in his work on higher-order projection analysis. Though more work is needed, it appears that this device will also make the PER theory more elegant.

Acknowledgements

Some parts of this paper are based closely on a previous joint paper with Kubiak and Hughes (Kubiak *et al.*, 1992), except that the material has been reformulated in terms of partial projections. The bulk of this paper was written while the authors were at Glasgow University.

References

Abramsky, S. (1990) *The Lazy Lambda Calculus*, in D. Turner editor, Declarative Programming. Reading, MA: Addison-Wesley.

Burn, G. L., Hankin, C. L. and Abramsky, S. (1986) Strictness Analysis for Higher-Order Functions, *Science of Computer Programming*, **7**.

Burn, G. L. (1990) A relationship between abstract interpretation and projection analysis, in *Proc. ACM Symposium on Principles of Programming Languages (POPL '90)*. New York: ACM Press.

Davis, K. (1993) Higher-order binding-time analysis, in *Proc. ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '93)*, New York: ACM Press.

Davis, K. (1994) Projection-based program analysis, PhD thesis, Glasgow University, UK.

Hughes, R. J. M. (1985) Strictness detection in non-flat domains, in H. Ganzinger and N. Jones, editors, *Proc. Workshop on Programs as Data Objects: LNCS 217*, Copenhagen, Denmark. Berlin: Springer-Verlag.

Hughes, R. J. M. (1988) Backwards analysis of functional programs, in Bjørner, Ershov and Jones, editors, *Partial Evaluation and Mixed Computation: Proc. IFIP TC2 Workshop*, Denmark. Amsterdam: North-Holland.

Hughes, R. J. M. and Launchbury, J. (1991) Towards relating forwards and backwards analyses, in S. L. Peyton Jones *et al.*, editors, *Functional Programming: Proc. Glasgow Workshop on Functional Programming*, pp. 13–15, Ullapool, Scotland. Berlin: Springer-Verlag.

Hughes, R. J. M. and Launchbury, J. (1992) Projections for polymorphic first-order strictness analysis. *Math. Struct. in Comp. Science, Vol. 2.* Cambridge: CUP.

Hunt, S. and Sands, D. (1991) Binding time analysis: a new PERspective, in *Proc. ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '91). ACM SIGPLAN Notices*, **26**(9).

Hunt, S. (1991) Abstract interpretation of functional languages: from theory to practice. PhD thesis, Imperial College, London, UK.

Jones, N. D. and Mycroft, A. (1986) Data flow analysis of applicative programs using minimal function graphs, in *Proc. 13th ACM Symposium on Principles of Programming Languages*, pp. 296–306, St. Petersburg, FL.

Johnsson, T. (1981) Detecting when call-by-value can be used instead of call- by-need. Prgramming Methodology Group Memo PMG-14, Institutionen för Informationsbehandling, Chalmers Tekniska Högskola, Göteborg, Sweden.

Kubiak, R., Hughes, R. J. M. and Launchbury, J. (1992) Implementing Projection-based Strictness Analysis. Department of Computing Science, University of Glasgow.

Launchbury, J. (1987) Projections for specialisation, in Bjørner, Ershov and Jones, editors, *Partial Evaluation and Mixed Computation: Proc. IFIP TC2 Workshop*, Denmark. Amsterdam: North-Holland.

Launchbury, J. (1991a) Projection factorisations in partial evaluation. PhD thesis, Glasgow University. (*Distinguished Dissertations in Computer Science*, Vol 1, Cambridge: CUP.)

Launchbury, J. (1991b) Strictness and Binding-time analyses: Two for the price of one, in *Proc. ACM Conference on Programming Language Design and Implementation.* New York: ACM Press.

Ong, C.-H. L. (1988) The lazy lambda calculus: an investigation in the foundations of functional programming PhD Thesis, Imperial College, London, UK.

Plotkin, G. D. (1985) *Lecture Notes.*

Peyton Jones, S.-L. and Launchbury, J. (1991) Unboxed Values as First Class Citizens in a Non-strict Functional Language. Department of Computing Science, University of Glasgow.

Schmidt, D. A. (1986) *Denotational Semantics.* MA: Allyn and Bacon.

Wadler, P. and Hughes, R. J. M. (1987) Projections for strictness analysis, in *Functional Programming and Computer Architecture: LNCS 274*, Portland, OR. Berlin: Springer-Verlag.

Wray, S. (1985) A new strictness detection algorithm, in *Proc. Workshop on Implementation of Functional Languages*, Aspenäs, Sweden.