Implementing Projection-based Strictness Analysis

Ryszard Kubiak, John Hughes, John Launchbury
Department of Computing Science
University of Glasgow

Abstract

Projection-based backwards strictness analysis has been understood for some years. Surprisingly, even though the method is fairly simple and quite general, no reports of its implementation have appeared. This paper describes ideas underlying our prototype implementation of the analysis for a simple programming language. The implementation serves as a case study before applying the method in the Glasgow Haskell compiler.

1 Introduction

The method of projection-based backwards strictness analysis for first-order, lazy functional languages was first presented by Wadler and Hughes [8] in 1987. Since then it has been generalised by Hughes [4] and Hughes and Launchbury [3] to work for user-defined types and for polymorphism. Yet, to our knowledge, it has never been implemented even though the method is fairly simple and quite general. The time has come for projection-based strictness analysis to meet practice.

This paper describes a prototype implementation. Initially we expected that building the prototype would involve little more than a routine application of the theory developed over the last few years. We were wrong. Many difficulties arose which corresponded either to loose ends, or to issues that were not explored in previous papers. Specifically, these involved the combination of polymorphism and user-defined types. To overcome these problems, we had to spend considerable effort tying the theory down sufficiently for its implementation to become practicable. The prototype is based on a heavily cut down version of Haskell, as our ultimate aim is to incorporate a projection-based strictness analysis phase in the Glasgow Haskell compiler. In particular, we restrict ourselves to a first-order polymorphic language.

Another reason for expecting the implementation to be straightforward was that an implementation of projection-based program analysis already existed. In his thesis [7], Launchbury implemented a *binding-time analysis* using projections, which analysed programs written in a polymorphic, first-order language with user defined types. In particular, he demonstrated that projections (which are functions) can effectively be manipulated in a concrete implementation. Unfortunately, strictness analysis involves additional complications which meant that the extensions to the binding-time analyser were not trivial.

In this paper we begin with a brief review of the method of projection-based strictness analysis. Then we introduce our example language, with its model for types and its semantics. We provide an abstract semantics based on projections which defines a strictness analysis. Implementation issues follow. We provide a generic method of generating finite lattices of projections (called *contexts*) over user-defined data types, and show how to model certain operations by manipulating their concrete representations. Finally, after covering more details of the implementation and giving some examples, we discuss the difficulties we face in extending it to Haskell.

The contributions this paper makes may be summarised as follows.

- We provide an explicit closure semantics for our example language, and give a new presentation of the projection analysis.
- We provide a complete set of rules for constructing finite domains of projections over polymorphic ground types, and describe how operations on these may be performed by an implementation.
- We provide a short suite of examples of the analyser at work, showing how the analysis of polymorphism and data structures interacts.

2 Projection-based Strictness Analysis

Early strictness analysis methods could discover nothing informative about functions on lazy data-structures, and projection-based strictness analysis was developed in an attempt to solve this problem. Recall that, in domain theory, a projection is a function p such that $p \circ p = p$ and $p \sqsubseteq \mathit{Ide}$ $(= \lambda x.x)$. The essential intuition is that a projection performs a certain amount of evaluation of a lazy data-structure. For example, the projection

$$\begin{array}{lcll} \multicolumn{4}{l}{\mathit{Left} : Nat \times Nat \rightarrow Nat \times Nat} \\ \mathit{Left}\ (x, y) & = & (-,-) & \textbf{if } x = - \\ & = & (x, y) & \textbf{if } x \neq - \end{array}$$

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

$$\begin{array}{lcll} \multicolumn{4}{l}{\mathit{Both} : Nat \times Nat \rightarrow Nat \times Nat} \\ \mathit{Both}\ (x, y) & = & (-,-) & \textbf{if } x = - \textbf{ or } y = - \\ & = & (x, y) & \textbf{otherwise} \end{array}$$

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 $+ : \mathit{Nat} \times \mathit{Nat} \rightarrow \mathit{Nat}$ 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.

$$\begin{array}{l} \mathit{swap} : Nat \times Nat \rightarrow Nat \times Nat \\ \mathit{swap}\ (x, y) = (y, x) \end{array}$$

While *swap* is not *Both*-strict, but 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 p and q be projections. We say f is p-strict in a q-strict context if $q \circ f = q \circ f \circ p$ (or equivalently, $q \circ f \sqsubseteq f \circ p$). $\square$

Projections capture the notion of evaluating a component of a data-structure. To capture evaluation of a single value we must embed it in a "data-structure" with a single component, which we can think of as representing an unevaluated closure. Thus we think of a closure of type t as an element of $t_\bot$, and we "evaluate" it with the projection

$$\begin{array}{lcll}
\mathit{Str} : t_\bot \rightarrow t_\bot & & & \\
\mathit{Str}\ \ - & = & - & \\
\mathit{Str}\ (\mathit{lift}\ x) & = & - & \textbf{if } x = - \\
 & = & \mathit{lift}\ x & \textbf{if } x \neq -
\end{array}$$

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

$$f \text{ is strict if and only if } \mathit{Str} \circ f_\bot \sqsubseteq f_\bot \circ \mathit{Str}$$

From f, together with a projection q representing the demand for the result of f, we want to find an p such that $q \circ f_\bot \sqsubseteq f_\bot \circ p$. We can always choose p to be *Ide*, but this is uninformative: *Ide* corresponds to performing no evaluation at all. We would like to find the smallest p such that the condition holds. In general this is equivalent to the halting problem, but [8] gives methods for finding quite small ps, for monomorphic functions. Later in this paper we give a revised version of these methods.

There are four fundamental projections over lifted types which capture various degrees of evaluation. We have already seen *Ide* (no evaluation), and *Str* (evaluate). In addition there is *Abs* defined by

$$\begin{array}{lcl}
\mathit{Abs} : t_\bot \rightarrow t_\bot & & \\
\mathit{Abs}\ \ - & = & - \\
\mathit{Abs}\ (\mathit{lift}\ x) & = & \mathit{lift}\ -
\end{array}$$

and the constant bottom function *Bot* $(= \lambda x.-)$. These are discussed in more detail later.

3 The Language

The language we use for the prototype implementation is a polymorphic, first-order lazy functional language with user-defined data types.

3.1 Syntax

Programs consist of type definitions, then function definitions, and finally an expression giving the meaning of the program. Each function definition and final expression must be given explicit type information. An example program is

```
List a = Nil + Cons a (List a)

append:: List a -> List a -> List a;
append xs ys = case xs
               in Nil -> ys
               || Cons u us -> Cons u (append us ys)
               end;

append (Cons Nil Nil) Nil:: List (List a)
```

In the abstract syntax which follows, we use {*pattern*} to signify zero or more repetitions.

Prog	$\rightarrow$	{TypeDef} {FnType FnDef} e::t
TypeDef	$\rightarrow$	T {a} = Sum
FnType	$\rightarrow$	f::{t ->}t
FnDef	$\rightarrow$	f {x} = e;
e	$\rightarrow$	x
	|	c {e}
	|	f {e}
	|	case e in c_1 {x_1}-> e_1 ...c_n {x_n}-> e_n end
Sum	$\rightarrow$	c_1 {t_1}+ ... +c_n {t_n}
t	$\rightarrow$	T {t}
	|	a

The grammar uses the following (possibly indexed) variables to denote the elements in various syntactic classes.

e	$\in$	*Expr*	[Value Expressions]
x	$\in$	*Var*	[Value Variables]
f	$\in$	*Fname*	[Function names]
c	$\in$	*Cname*	[Constructor names]
t	$\in$	*Texpr*	[Type Expressions]
a	$\in$	*Tvar*	[Type variables]
T	$\in$	*Tname*	[Type names]

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,

```
type Pair a b = MkPair a b;

fst:: Pair a b -> a;
fst z = case z in MkPair x y -> x end;
```

The reason for this omission is to remove confusion between pairing in the language, and product in the semantics. The former is modelled by a lifted product.

3.2 Semantics

In order to use *Str* to discover simple strictness, the methods of [8] demand that instead of analysing a function $f : A \to B$, its lifted version $f_\perp : A_\perp \to B_\perp$ is used instead. At first this seems to be just a technical trick. However, lifting plays a role both in theory and in practice for modelling lazy evaluation.

Non-strict semantics requires that the evaluation of an argument to a function call should be postponed until it is certain the argument is needed. In practice, when evaluating a function call, the arguments are stored in the form of *closures*, i.e. graphs representing the expressions, and it is only evaluation of the function's body which causes evaluation of the closures. We model closures by lifted values.

3.2.1 Denotations of Types

We depart slightly from the standard model of types in lazy functional languages. Usually the right hand side of a type definition of the form

```
F a b = C1 R S + C2 T
```

is modelled by the domain

$$R \times S \;+\; T$$

where R, S and T model `R`, `S` and `T` respectively, the product is cartesian product, and the sum is separated sum. Instead, we model the type by

$$R_\perp \otimes S_\perp \;\oplus\; T_\perp$$

using smash sum and smash product. Because of the well-known isomorphisms $R + S \cong R_\perp \oplus S_\perp$ and $(R \times S)_\perp \cong R_\perp \otimes S_\perp$ this is isomorphic to the usual, but it is more convenient for us as it allows us to make explicit where closures reside. As an example, consider lists, defined as follows.

```
List a = Nil + Cons a (List a);
```

We model this type by the domain (actually functor),

$$\mathit{List} = \Lambda\alpha \;.\; \mu L \;.\; \mathbf{1}_\perp \;\oplus\; (\alpha_\perp \otimes L_\perp)$$

We write $\mathit{in}^s_{\mathit{Nil}}$ and $\mathit{in}^s_{\mathit{Cons}}$ for the injection functions into a named smash sum. Normally, we will drop the explicit $\Lambda\alpha$, and simply use successive Greek letters for successive polymorphic parameters (a sort of de Bruijn index).

3.2.2 Dynamic Semantics

The semantics are given in terms of two semantic functions,

$$\begin{aligned}&\mathcal{PR} : \mathit{FunDefs} \to \mathit{FunEnv}\\ &\mathcal{E} : \mathit{FunEnv} \to \mathit{Expr} \to \mathit{ValEnv} \to \mathit{Value}\end{aligned}$$

where

$$\begin{array}{lcl} \nu \in \mathit{Value} & = & \bigcup_{\tau \in \mathrm{Type}} \tau \\ \rho \in \mathit{ValEnv} & = & \mathit{Var} \to \mathit{Value}_\perp \\ \phi \in \mathit{FunEnv} & = & \mathit{Fname} \to (\mathit{Value}_\perp \otimes \cdots \otimes \mathit{Value}_\perp \to \mathit{Value}) \end{array}$$

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. $\mathcal{E}$ is defined as follows.

$$\begin{array}{lcl} \mathcal{E}_\phi[\![\,\mathtt{x}\,]\!]_\rho & = & \mathit{drop}\ (\rho\ (\mathtt{x})) \\[1em] \mathcal{E}_\phi[\![\,\mathtt{f}\ \mathtt{e}_1 \ldots \mathtt{e}_k\,]\!]_\rho & = & \phi\ (\mathtt{f})\ (\mathit{lift}\ \mathcal{E}_\phi[\![\,\mathtt{e}_1\,]\!]_\rho \otimes \cdots \otimes \mathit{lift}\ \mathcal{E}_\phi[\![\,\mathtt{e}_k\,]\!]_\rho) \\[1em] \mathcal{E}_\phi[\![\,\mathtt{c}\ \mathtt{e}_1 \ldots \mathtt{e}_k\,]\!]_\rho & = & \mathit{in}^s_{\mathtt{C}}\ (\mathit{lift}\ \mathcal{E}_\phi[\![\,\mathtt{e}_1\,]\!]_\rho \otimes \cdots \otimes \mathit{lift}\ \mathcal{E}_\phi[\![\,\mathtt{e}_k\,]\!]_\rho) \end{array}$$

$$\begin{array}{lcl} \mathcal{E}_\phi[\![\,\mathtt{case}\ \mathtt{e}\ \mathtt{in}\ \cdots \mathtt{c}_j\ \mathtt{x}_1 \ldots \mathtt{x}_k\ \mathtt{->}\ \mathtt{e}_j \cdots]\!]_\rho & & \\ \quad = & \mathit{case}\ \mathcal{E}_\phi[\![\,\mathtt{x}\,]\!]_\rho\ \mathit{in} & \\ & \quad - \quad \to \quad - & \\ & \quad \vdots & \\ & \mathit{in}^s_{\mathtt{C}_j}\ (\nu_1 \otimes \cdots \otimes \nu_k) \ \to \ \mathcal{E}_\phi[\![\,\mathtt{e}_j\,]\!]_{\rho[\mathtt{x}_i \mapsto \nu_i]_i} & \\ & \quad \vdots & \end{array}$$

Note our overloading of the symbol $\otimes$, here to operate on values. Any ambiguity about the use of such overloaded notation can be resolved by the context.

The novel aspect of the definition of $\mathcal{E}$ is the use of explicit closures. For example, when a variable is dereferenced, it's closure is collapsed (i.e. it is evaluated) using the function $\mathit{drop} : t_\perp \to t$ which maps $-$ to $-$, and $\mathit{lift}\ x$ to x. Conversely, when a function is called, a tuple of closures is constructed strictly containing its arguments, and so on. This is an accurate simulation of what actually happens within the machine: if for any reason the tuple of closures for the arguments cannot be created (e.g. the lack of memory) the function returns $-$. While the definition of $\mathcal{E}$ looks unusual, it may be obtained from the usual semantics via the isomorphisms of Section 3.2.1.

The function environment is constructed as the least fixed point of the function definitions, as follows (where $\underline{\lambda}$ produces a strict function),

$$\begin{aligned}&\mathcal{PR}[\![\cdots,\ \mathtt{f}\ \mathtt{x}_1\ \mathtt{..}\ \mathtt{x}_k\ \mathtt{=}\ \mathtt{e},\ \cdots]\!]\\ &\quad = \mathit{fix}\ (\lambda\phi.\{\cdots, f \mapsto \underline{\lambda}(\nu_1 \otimes \cdots \otimes \nu_k).\mathcal{E}_\phi[\![\,\mathtt{e}\,]\!]_{[\mathtt{x}_i \mapsto \nu_i]}, \cdots\})\end{aligned}$$

3.3 Non-uniform Types

The syntax of types allowed by most lazy functional languages (including the language used in this paper) allows for *non-uniformly* recursive types. A type is *uniformly-recursive* if it may be defined by equations of the form $\mu X.F(X)$, where F is a functor from domains to domains (i.e. $F(X)$ is a type-valued expression depending on X).

The following are two examples of non-uniform types.

```
type Moo a b = Msimple + Mcompl (Moo b a);
type Foo t = Fsimple + Fcompl (Foo (Foo t));
```

To model such types requires abstraction over functors, rather than just over types. Furthermore, while fairly trivial programs containing non-uniform types can be successfully type-checked using the widespread Hindley-Milner type system, the obvious versions of `map` and `fold` for such types cannot.

If we allowed them, non-uniform types would also cause problems for us later on when we define lattices of contexts over each type. For these reasons we rule out non-uniform types. This may be done with a syntactic check.

It is worth mentioning that the restriction does not exclude definitions such as

```
type Goo a = Gsimple + Gcompl (List (Goo a))
```

in which `Goo a` is the instantiation parameter for `List`.

4 Abstract Semantics

In this section we define an abstract semantics for our language to perform backward strictness analysis. It takes the form of a projection transformer.

Wadler and Hughes introduced two operations for combining projections: $\sqcup$ and &. The first is usual least upper bound (i.e. pointwise). The second is defined as follows.

$$\begin{array}{lcll}(p\&q)\ x & = & - & \textbf{if}\ p\ x = -\ \textbf{or}\ q\ x = - \\ & = & (p \sqcup q)\ x & \textbf{otherwise}\end{array}$$

This operation is used in the analysis to capture conjunction of demand. In our presentation, we extend these operations to denote corresponding pointwise operations on abstract environments. In such an environment

$$env \ \in \ AbsEnv = Var \rightarrow Proj$$

names are associated with projections over lifted types. We use [ ] to denote the initial environment in which every identifier is mapped to *Bot*. By $[(\mathtt{x}, p)]$ we mean the initial environment extended by binding the variable x to the projection p, 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 *Bot*.

A forwards projection analysis such as appears in Launchbury's thesis [7] is defined by abstract functions which mimic the concrete semantic functions.

$$\begin{array}{l}\mathcal{PR}^{\#} : FunDefs \rightarrow ForwardAbsFunEnv \\ \mathcal{E}^{\#} : ForwardAbsFunEnv \rightarrow Expr \rightarrow (AbsValEnv \rightarrow AbsValue)\end{array}$$

In the case of backwards strictness analysis the direction of the final arrow is reversed.

$$\begin{array}{l} \mathcal{PR}^{\#} : \mathit{FunDefs} \rightarrow \mathit{AbsFunEnv} \\ \mathcal{E}^{\#} : \mathit{AbsFunEnv} \rightarrow \mathit{Expr} \rightarrow (\mathit{AbsValue} \rightarrow \mathit{AbsValEnv}) \end{array}$$

where,

$$\begin{array}{rcl} p \ \in \ \mathit{AbsValue} & = & \text{some domain of projections} \\ \rho^{\#} \ \in \ \mathit{AbsValEnv} & = & \mathit{Var} \rightarrow \mathit{AbsValue} \\ \phi^{\#} \ \in \ \mathit{AbsFunEnv} & = & \mathit{Fname} \rightarrow (\mathit{AbsValue} \rightarrow \mathit{AbsValEnv}) \end{array}$$

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

$$\begin{array}{l} p \quad \circ \quad \underline{\lambda}(\nu_1 \otimes \cdots \otimes \nu_k).\mathit{lift}\ \mathcal{E}_{\phi}[\![\,\mathtt{e}\,]\!]_{[\mathtt{x}_i \mapsto \nu_i]} \\ \qquad\qquad \sqsubseteq\ \underline{\lambda}(\nu_1 \otimes \cdots \otimes \nu_k).\mathit{lift}\ \mathcal{E}_{\phi}[\![\,\mathtt{e}\,]\!]_{[\mathtt{x}_i \mapsto \nu_i]} \quad \circ \quad (\rho(\mathtt{x}_1) \otimes \cdots \otimes \rho(\mathtt{x}_k)) \end{array}$$

The environment $\rho(\mathtt{x}_i)$ is the demand on parameter $\mathtt{x}_i$ given a demand p on the value of $\mathtt{e}$, 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.

To capture strictness properties we will re-express the projection *Str* using a strict form of the lifting operation, written $(-)_{\oplus}$. On domains, $t_{\perp} = t_{\oplus}$, but on functions,

$$\begin{array}{lcll} f_{\oplus}\ \ - & = & - & \\ f_{\oplus}\ (\mathit{lift}\ x) & = & - & \textbf{if}\ f\ x = - \\ & = & \mathit{lift}\ (f\ x) & \textbf{otherwise} \end{array}$$

Strict lifting is functorial except that it doesn't preserve the identity. In fact, $\mathit{Ide}_{\oplus} = \mathit{Str}$. Writing $f_{\oplus}$ provides a more convenient and compact notation for $(\mathit{Str} \circ f_{\perp})$, as occurs in earlier papers.

Our first use of strict lifting is in the definition of the projection transformer $\mathcal{E}^{\#}$. The first equation realises the guard operation from [8].

$$\mathcal{E}^{\#}_{\phi^{\#}}[\![\,\mathtt{e}\,]\!](p_{\perp}) \quad = \quad \mathcal{E}^{\#}_{\phi^{\#}}[\![\,\mathtt{e}\,]\!](p_{\oplus}) \quad \sqcup \quad \lambda x.\mathit{Abs}$$

Recall that all demands are projections over lifted domains. Such projections may always be expressed in the form either $p_{\perp}$ or $p_{\oplus}$. The intuition behind a demand of the form $p_{\perp}$ is, "this value may or may not be required, but if it is then p's worth will be needed." Conversely, a demand of the form $p_{\oplus}$ means, "this value *will* be required, and what's more, p'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." Note that *Abs* is the weakest lazy demand, as $\mathit{Abs} = \mathit{Bot}_{\perp}$.

The rest of the equations apply to projections expressible as $p_{\oplus}$

$$\mathcal{E}^{\#}_{\phi^{\#}}[\![\,\mathtt{x}\,]\!]p \;=\; [(\mathtt{x}, p)]$$

$$\begin{aligned}
&\mathcal{E}^{\#}_{\phi^{\#}}[\![\,\mathtt{f}\ \mathtt{e}_1 \ldots \mathtt{e}_k\,]\!]p_{\oplus} \\
&\qquad = \mathcal{E}^{\#}_{\phi^{\#}}[\![\,\mathtt{e}_1\,]\!]p_1 \;\&\cdots\&\; \mathcal{E}^{\#}_{\phi^{\#}}[\![\,\mathtt{e}_k\,]\!]p_k \\
&\qquad\quad \textbf{where } (p_1 \otimes \cdots \otimes p_k) = (\phi^{\#}\ f)\ p
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}^{\#}_{\phi^{\#}}[\![\,\mathtt{c}\ \mathtt{e}_1 \ldots \mathtt{e}_k\,]\!](\cdots \oplus (p_1 \otimes \cdots \otimes p_k) \oplus \cdots)_{\oplus} \\
&\qquad = \mathcal{E}^{\#}_{\phi^{\#}}[\![\,\mathtt{e}_1\,]\!]p_1 \;\&\cdots\&\; \mathcal{E}^{\#}_{\phi^{\#}}[\![\,\mathtt{e}_k\,]\!]p_k
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}^{\#}_{\phi^{\#}}[\![\,\mathtt{case\ e\ in}\ \cdots \mathtt{c}_i\ \mathtt{x}_1 \ldots\ \mathtt{x}_k\mathtt{->e}_i \cdots\ \mathtt{end}\,]\!]p_{\oplus} \\
&\qquad = \textstyle\bigsqcup_i \left(\mathcal{E}^{\#}_{\phi^{\#}}[\![\,\mathtt{e}\,]\!](\delta_i)_{\oplus} \;\&\; \rho_i \setminus \{\mathtt{x}_1, \ldots, \mathtt{x}_k\}\right) \\
&\qquad\quad \textbf{where} \\
&\qquad\qquad \rho_i = \mathcal{E}^{\#}_{\phi^{\#}}[\![\,\mathtt{e}_i\,]\!]p_{\oplus} \\
&\qquad\qquad \delta_i = \mathit{Bot} \oplus \cdots \oplus (\rho_i\ \mathtt{x}_1 \otimes \cdots \otimes \rho_i\ \mathtt{x}_k) \oplus \cdots \oplus \mathit{Bot}
\end{aligned}$$

In these rules we assume that the structure of the contexts corresponding 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 assumed to be the one associated with the constructor `c`. Similarly, the *Bot*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`.

It is worth noticing that our transformer is slightly more efficient than the one given originally [8]. Like Davis and Wadler [2], we collect the result of the analysis in an environment of projections, so allowing the contexts for all the arguments to be found by a single pass through the function's body.

Because of the first equation for $\mathcal{E}^{\#}$ we only need to construct the function environment for strict demands. Again it is constructed as the least fixed point of the function definitions.

$$\begin{aligned}
&\mathcal{PR}^{\#}[\![\cdots,\ \mathtt{f}\ \mathtt{x}_1\ ..\ \mathtt{x}_k\ \mathtt{=}\ \mathtt{e},\ \cdots]\!] \\
&\quad = \mathit{fix}\ (\lambda\phi^{\#}.\{\cdots, f \mapsto \lambda p\ .\ \mathcal{E}^{\#}_{\phi^{\#}}[\![\,\mathtt{e}\,]\!]\ (p_{\oplus}), \cdots\})
\end{aligned}$$

5 Manipulating Projections

Projections are functions, yet our analyser needs to perform a variety of operations on them, including comparing for equality. To achieve this we work with a concrete representation of the projections, syntactically modelling the semantic constructions and operations. This is reflected in the following development where we use the semantic notation to represent a syntactic construction.

5.1 Contexts for strictness analysis

Following the approach of Launchbury [6] and Hughes and Launchbury [3] we define for each type a finite collection of projections called *contexts*. The following rules define the family of all contexts by induction on the structure of the denotations of types.

$$\mathbf{1}\ \mathbf{cxt}\ \mathbf{1} \qquad \alpha\ \mathbf{cxt}\ \alpha$$

$$\frac{p\ \mathbf{cxt}\ T}{p_{\bot}\ \mathbf{cxt}\ T_{\bot}} \qquad \frac{p\ \mathbf{cxt}\ T}{p_{\obot}\ \mathbf{cxt}\ T_{\bot}}$$

$$\frac{p_1\ \mathbf{cxt}\ T_1\ \ldots\ p_n\ \mathbf{cxt}\ T_n}{p_1 \oplus \ldots \oplus p_n\ \mathbf{cxt}\ T_1 \oplus \ldots \oplus T_n}$$

$$\frac{p_1\ \mathbf{cxt}\ T_1\ \ldots\ p_n\ \mathbf{cxt}\ T_n}{p_1 \otimes \ldots \otimes p_n\ \mathbf{cxt}\ T_1 \otimes \ldots \otimes T_n}$$

$$\frac{P(p)\ \mathbf{cxt}\ T(t) \qquad [p\ \mathbf{cxt}\ t]}{\mu p.P(p)\ \mathbf{cxt}\ \mu t.T(t)}$$

$$\frac{p\ \mathbf{cxt}\ F \quad q_1\ \mathbf{cxt}\ T_1\ \ldots\ q_n\ \mathbf{cxt}\ T_n}{p\ q_1 \ldots q_n\ \ \mathbf{cxt}\ \ F\ T_1 \ldots T_n}$$

As an example of contexts we present the familiar head-strict and tail-strict projections [8] over the the list type $\mu L\ .\ \mathbf{1}_{\bot}\ \oplus\ \alpha_{\bot} \otimes L_{\bot}$.

$$\begin{array}{l} H = \mu l\ .\ \mathbf{1}_{\bot}\ \oplus\ \alpha_{\obot} \otimes l_{\bot} \\ T = \mu l\ .\ \mathbf{1}_{\bot}\ \oplus\ \alpha_{\bot} \otimes l_{\obot} \end{array}$$

Strictly speaking, the polymorphic projections are represented by $(H\ Ide)$ and $(T\ Ide)$, but we will often be sloppy and understand that uninstantiated parameters α, β etc. are actually instantiated to Ide.

5.2 Modelling Operations

The analysis is defined in terms of operations on projections. In order to implement the analysis we need to model these semantic operations on contexts. To help us do this, we need to explore some of the properties of the operations.

First, the least upper bound operation $\sqcup$. If p, q are projections then $p \sqcup q$ defined point-wise is also a projection. Furthermore, the following equalities hold.

$$\begin{array}{lcl} p \sqcup p & = & p \\ p \sqcup q & = & q \sqcup p \\ (p \sqcup q) \sqcup r & = & p \sqcup (q \sqcup r) \\ p \sqcup Bot & = & p \\ p \sqcup Ide & = & Ide \\ p_{\bot} & = & Abs \sqcup p_{\obot} \\ (p \oplus q) \sqcup (r \oplus s) & = & (p \sqcup r) \oplus (q \sqcup s) \\ (p \otimes q) \sqcup (r \otimes s) & = & (p \sqcup r) \otimes (q \sqcup s) \\ p_{\bot} \sqcup q_{\bot} & = & (p \sqcup q)_{\bot} \\ (p_{\obot}) \sqcup q_{\bot} & = & (p \sqcup q)_{\bot} \\ (p_{\obot}) \sqcup (q_{\obot}) & = & (p \sqcup q)_{\obot} \\ \mu p.P(p) \sqcup \mu q.Q(q) & = & \mu p.(P \sqcup Q)(p) \end{array}$$

All these but the last can be easily checked from appropriate definitions. The last rule requires induction over the context structure of P and Q. The equalities in the distributivity rules guarantee that if p and q are contexts, then $p \sqcup q$ is a context too.

Unfortunately, although contexts are closed under $\sqcup$ they are not closed under &, and so we may need to introduce extra approximation. The following properties of & may be derived straight from its definition.

$$\begin{array}{lcl}
p \,\&\, p & = & p \\
p \,\&\, q & = & q \,\&\, p \\
(p \,\&\, q) \,\&\, r & = & p \,\&\, (q \,\&\, r) \\
(p \sqcup q) \,\&\, r & = & (p \,\&\, r) \sqcup (q \,\&\, r) \\
p \,\&\, \mathit{Bot} & = & \mathit{Bot} \\
p_\perp \,\&\, \mathit{Abs} & = & p_\perp \\
\mathit{Str} \,\&\, \mathit{Ide} & = & \mathit{Str} \\
p_{\oplus\!\!\!\perp} \,\&\, q_{\oplus\!\!\!\perp} & = & (p \,\&\, q)_{\oplus\!\!\!\perp} \\
p_{\oplus\!\!\!\perp} \,\&\, q_\perp & = & (p \;\sqcup\; (p \,\&\, q))_{\oplus\!\!\!\perp} \\
p_\perp \,\&\, q_\perp & = & (p \sqcup q)_\perp \\
(p \oplus q) \,\&\, (r \oplus s) & = & (p \,\&\, r) \oplus (q \,\&\, s) \\
(p \otimes q) \,\&\, (r \otimes s) & = & (p \,\&\, r) \otimes (q \,\&\, s)
\end{array}$$

While the & operation distributes nicely over products and sums it is not the case with the fixed-points. This means $\mu p.P(p)\&\mu q.Q(q)$ is not necessarily equal to $\mu p.(P\&Q)(p)$. The familiar projections H and T over lists provide us with an example of this, that is, $H\&T \neq \mu l\ .\ \mathbf{1}_\perp \ \oplus\ \mathit{Ide}_{\oplus\!\!\!\perp} \otimes l_{\oplus\!\!\!\perp}$.

To see this, consider applying each to the list $u = \mathit{1} : - : [\,]$. We see that $T\ \ u = \mathit{1} : - : [\,]$ and $H\ \ u = \mathit{1} : -$. As neither returns $-$,

$$(H\&T)\ u = (H\ \ u) \sqcup (T\ \ u) = \mathit{1} : - : [\,]$$

However, $(\mu l\ .\ \mathbf{1}_\perp \ \oplus\ \mathit{Ide}_{\oplus\!\!\!\perp} \otimes l_{\oplus\!\!\!\perp})\ u = -$ and so it is not equal to $H\&T$.

This apparently peculiar behaviour of H and T comes from the fact that T is strict over its recursive calls while H is not. The actual projection $H\&T$ treats the head of its argument differently from the tail, and so is not a context. Because we want to disallow such projections in order to retain finite domains, we need a way to find the least context approximating $H\&T$ from above. We may start from unfolding H and T in $H\&T$.

$$\begin{array}{lcl}
H\&T & = & (\mathbf{1}_\perp \ \oplus\ \mathit{Ide}_{\oplus\!\!\!\perp} \otimes H_\perp) \,\&\, (\mathbf{1}_\perp \ \oplus\ \mathit{Ide}_\perp \otimes T_{\oplus\!\!\!\perp}) \\
 & = & \mathbf{1}_\perp \ \oplus\ \mathit{Ide}_{\oplus\!\!\!\perp} \otimes (T \ \sqcup\ H\&T)_{\oplus\!\!\!\perp}
\end{array}$$

This shows that the desired approximation to $H\&T$ cannot be less than $T \ \sqcup\ H\&T$. We may continue with unfolding the latter

$$\begin{array}{l}
T \sqcup H\&T \\
\quad = \ (\mathbf{1}_\perp \oplus \mathit{Ide}_\perp \otimes (T_{\oplus\!\!\!\perp}))\ \sqcup (\mathbf{1}_\perp \oplus \mathit{Ide}_{\oplus\!\!\!\perp} \otimes (T \sqcup H\&T)_{\oplus\!\!\!\perp}) \\
\quad = \ \mathbf{1}_\perp \oplus \mathit{Ide}_\perp \otimes (T \sqcup H\&T)_{\oplus\!\!\!\perp}
\end{array}$$

We obtain a recursive equation with respect to $T \sqcup H\&T$ to which the minimal solution is the context $\mu l\ .\ \mathbf{1}_\perp \ \oplus\ \mathit{Ide}_{\oplus\!\!\!\perp} \otimes l_{\oplus\!\!\!\perp}$. This is actually T which is therefore the least context above $H\&T$.

Let us find out what can be the best approximation to $\mu p.P(p)\&\mu q.Q(q)$ in general. To ease the notation we will write $\mu p.P(p)$ as μP, and likewise for $\mu q.Q(q)$.

A closer look at the properties of &, especially those dealing with lifted projections, allow us to conclude that whenever $\mu P \ \& \ \mu Q$ is unfolded we will obtain one of the following combinations: $\mu P \ \& \ \mu Q$, $\mu P \ \sqcup \ \mu P\&\mu Q$, $\mu Q \ \sqcup \ \mu P\&\mu Q$ or $\mu P \sqcup \mu Q$. Which of these appear depends on which of the recursive calls inside μP and μQ are lifted with strict-lifting, and which with usual lazy lifting. There are four possible cases for the conjunction of the unfolded context:

- it depends only on $\mu P \ \& \ \mu Q$; in this case $\mu P \ \& \ \mu Q = \mu(P\&Q)$, the latter being a context (by induction), and we do not need to approximate;
- the conjunction contains $\mu P \sqcup \mu Q$; this is equal to the context $\mu(P \sqcup Q)$ and we cannot do better than taking this as the approximation;
- both $\mu P \ \sqcup \ \mu P\&\mu Q$ and $\mu Q \ \sqcup \ \mu P\&\mu Q$ appear in the result; a further unfolding of any of these will show the dependence on $\mu P \sqcup \mu Q$ and we again must take $\mu(P \sqcup Q)$ as the approximation; and
- only one of $\mu P \ \sqcup \ \mu P\&\mu Q$ or $\mu Q \ \sqcup \ \mu P\&\mu Q$ occurs (possibly together with $\mu P \ \& \ \mu Q$); a further unfolding will show that $\mu P \ \sqcup \ \mu P\&\mu Q$ or, respectively, $\mu Q \ \sqcup \ \mu P\&\mu Q$ is the best approximation.

The last case corresponds to our previous example $H\&T$ for which T is the best approximation.

Our implementation does not unfold $\mu P \ \& \ \mu Q$ when it evaluates its value. Instead, all the pairs of corresponding recursive calls inside μP and μQ are analysed. For each pair of calls only the information whether the call is lifted strictly or lazily is recorded. Assuming an ordering on pairs as in the picture

$$\begin{tikzcd}
 & (lazy, lazy) \arrow[dl, dash] \arrow[dr, dash] & \\
(strict, lazy) \arrow[dr, dash] & & (lazy, strict) \arrow[dl, dash] \\
 & (strict, strict) &
\end{tikzcd}$$

the implementation evaluates the least upper bound of all pairs and on this basis one of the four discussed cases applies. This approach requires a single traversal through the structure of μP and μQ in order to find the best approximation to $\mu P\&\mu Q$.

6 Implementation

6.1 Finding Fixed Points

A program containing recursive function definitions gives rise to recursive abstract equations, which have to be solved at compile time. Of the variety of methods for finding such solutions (fixed points) we use minimal function graphs [5]. Our implementation follows that of Launchbury [7].

The minimal function graph contains abstract function results (tuples of contexts representing the demand for the arguments to the function) for selected abstract arguments (contexts representing the demand for the function's result) for the functions in the program. The process of evaluating the fixed-point starts by constructing a graph containing the arguments we are interested in, along with an initial approximation for the function's result. The following iterative process is applied in order to improve these approximations.

At each iteration all argument-value pairs recorded in the current function graph are reevaluated according to the projection transformer. This is done by applying the transformer to the function body and the argument. When a function application is encountered within the body, two cases are possible depending on whether we can find the function value for an argument in the current graph. If we can, the value of the application is taken from the graph. If we cannot, we compute the best approximation derivable from the current graph, and add the new point to the graph.

At each iterative step we construct a fresh graph containing improved approximations to the values previously recorded, together with argument-value pairs for newly met arguments. With this new graph the iterative process continues until successive graphs are equal, that is until the fixed-point solution for the whole program is reached. As an optimisation to this process, we split the graph into mutually dependent units, and only iterate those parts that might have changed.

Minimal function graphs appear to be crucial for obtaining strictness analysis results in a practically acceptable time, given that the number of possible contexts over arbitrary data types can be quite large. The gain comes from the fact that we rarely require the whole fixed-point, but only its value for a few specified arguments.

6.2 Concrete Representation of Projections

The following is the LML [1] definition of the type that represents projections in the implementation

```
type proj = PStr proj
          + PLift proj
          + PBot
          + PProd (List proj)
          + PMu name (List proj)
          + PRec name
          + PSum (List proj)
```

The term `PLift p` corresponds to $\mathrm{p}_\perp$ and `PStr p` represents $\mathit{Str}\ o\ \mathrm{p}_\perp$. Data of the form `PProd [`$\mathrm{p}_1,\ldots,\mathrm{p}_k$`]`, `PSum [`$\mathrm{p}_1,\ldots,\mathrm{p}_k$`]` stand for strict products $\mathrm{p}_1 \otimes \ldots \otimes \mathrm{p}_k$ and sums $\mathrm{p}_1 \oplus \ldots \oplus \mathrm{p}_k$, respectively.

The structures `PBot` and `PProd []` represent the monomorphic bottom and identity projections over the two-point domain $\mathbf{1}_\perp$, as well as the polymorphic projections *Bot* and *Ide*, respectively. This overloading does not confuse us in practice because when needed we can deduce from a surrounding context what is the underlying domain for a projection.

A data of the form `PMu f [p`$_1$`,...,p`$_k$`]` represents a context over a recursive type. As we admit mutual recursion on types the representation allows for mutually recursive contexts `p`$_1$`,...,p`$_k$ over domains mutually recursive with `f`. Of course, among these contexts there should be a context over `f`. The `PRec g` form can only be met inside a recursive projection and the name `g` selects one of the surrounding mutually recursive projections.

On the `proj` type a library of basic operations is defined such as generating the bottom or the identity contexts for a given type, calculation of the $\sqcup$ and &, factorisation of an instance of a polymorphic projection into its polymorphic and instance parts. The abstract function graphs stores the argument-value pairs for abstract functions also as data of the type `proj`.

6.3 Sample Results

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

6.3.1 Lists

We begin with some standard list-based examples.

```
type List alpha = Nil + Cons alpha (List alpha);

append ::  List alpha -> List alpha -> List alpha;
append  xs zs  = case xs in
                    Nil -> zs
                 || Cons y ys  -> Cons y (append ys zs)
                 end;

reverse :: List alpha -> List alpha;
reverse rs = case rs in
                Nil -> Nil
             || Cons y ys -> append (reverse ys) (Cons y Nil)
             end;
```

First *append*. Consider the demand $(\mu l\ .\ \mathit{Nil} : \mathbf{1}_\bot\ \oplus\ \mathit{Cons} : \alpha_{\obot} \otimes l_\bot)_{\obot}$ for *append*'s result. This is a strict demand (hence the final strict lifting) which is recursively strict in each list element (hence the strict lifting on the α), but lazy in the list tails (the non-strict lifting of l). This is what we previously wrote as $H_{\obot}$. From this result context, the demand on *append*'s arguments is computed as,

$$(\mu l\ .\ \mathit{Nil} : \mathbf{1}_\bot\ \oplus\ \mathit{Cons} : \alpha_{\obot} \otimes l_\bot)_{\obot}\ \otimes\ (\mu l\ .\ \mathit{Nil} : \mathbf{1}_\bot\ \oplus\ \mathit{Cons} : \alpha_{\obot} \otimes l_\bot)_\bot$$

In summary, a strict, head-strict demand $H_{\obot}$ 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. $H_{\obot} \otimes H_\bot$.

Alternatively, given a demand $(\mu l\ .\ \mathit{Nil} : \mathbf{1}_\perp \ \oplus\ \mathit{Cons} : \alpha_\perp \otimes l_\phi)_\phi$ (that is, strict and tail-strict, T_ϕ) for the result of *append*, the analyser deduced a demand of

$$(\mu l\ .\ \mathit{Nil} : \mathbf{1}_\perp \ \oplus\ \mathit{Cons} : \alpha_\perp \otimes l_\phi)_\phi \ \otimes\ (\mu l\ .\ \mathit{Nil} : \mathbf{1}_\perp \ \oplus\ \mathit{Cons} : \alpha_\perp \otimes l_\phi)_\phi$$

for its arguments, i.e. both arguments strict and tail strict, $T_\phi \otimes T_\phi$.

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

$$(\mu l\ .\ \mathit{Nil} : \mathbf{1}_\perp \ \oplus\ \mathit{Cons} : \alpha_\phi \otimes l_\perp)_\phi$$

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

$$(\mu l\ .\ \mathit{Nil} : \mathbf{1}_\perp \ \oplus\ \mathit{Cons} : \alpha_\perp \otimes l_\phi)_\phi$$

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 H_ϕ and T_ϕ contexts.

6.3.2 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.

```
type Tree alpha = Leaf alpha + Node (Tree alpha) (Tree alpha);

flat :: Tree alpha -> List alpha;
flat t = case t in
             Leaf x   -> Cons x Nil
          || Node l r -> append (flat l) (flat r)
          end;
```

Whereas particular contexts over lists like H and T 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 *flat* collapses a tree down to a list. If that result list is demanded by a strict, and head-strict context, H_ϕ, that is,

$$(\mu l\ .\ \mathit{Nil} : \mathbf{1}_\perp \ \oplus\ \mathit{Cons} : \alpha_\phi \otimes l_\perp)_\phi$$

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

$$(\mu t\ .\ \mathit{Leaf} : \alpha_\phi \ \oplus\ \mathit{Node} : t_\phi \otimes t_\perp)_\phi$$

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 *flat* is demanded in a strict, and tail-strict context, $T_{\oplus}$, that is,

$$(\mu l\ .\ \mathit{Nil} : \mathbf{1}_{\perp}\ \oplus\ \mathit{Cons} : \alpha_{\perp} \otimes l_{\oplus})_{\oplus}$$

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

$$(\mu t\ .\ \mathit{Leaf} : \alpha_{\perp}\ \oplus\ \mathit{Node} : t_{\oplus} \otimes t_{\oplus})_{\oplus}$$

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

6.3.3 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.

```
type Nat = Zero + Succ Nat;

add :: Nat -> Nat -> Nat;
add a b = case a in Zero -> b || Succ c -> Succ (add c b) end;

sum :: Tree Nat -> Nat;
sum t = case t in
           Leaf t   -> t
        || Node l r -> add (sum l) (sum r)
        end;
```

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

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

produces an argument context of

$$(\mu n\ .\ \mathit{Zero} : \mathbf{1}_{\perp}\ \oplus\ \mathit{Succ} : n_{\oplus})_{\oplus}\ \otimes\ (\mu n\ .\ \mathit{Zero} : \mathbf{1}_{\perp}\ \oplus\ \mathit{Succ} : n_{\oplus})_{\oplus}$$

Conversely, a result context of

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

generates the argument context

$$(\mu n\ .\ \mathit{Zero} : \mathbf{1}_{\perp}\ \oplus\ \mathit{Succ} : n_{\perp})_{\oplus}\ \otimes\ (\mu n\ .\ \mathit{Zero} : \mathbf{1}_{\perp}\ \oplus\ \mathit{Succ} : n_{\perp})_{\perp}$$

In other words, if the complete result of *add* is demanded then both is 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,

$$(\mu n\ .\ Zero : \mathbf{1}_{\bot}\ \oplus\ Succ : n_{\obar})_{\obar}$$

is converted to the argument context,

$$((\mu t\ .\ Leaf : \alpha_{\obar}\ \oplus\ Node : t_{\obar} \otimes t_{\obar})\ (\mu n\ .\ Zero : \mathbf{1}_{\bot}\ \oplus\ Succ : n_{\obar}))_{\obar}$$

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.

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

$$(\mu n\ .\ Zero : \mathbf{1}_{\bot}\ \oplus\ Succ : n_{\bot})_{\obar}$$

to the argument context

$$((\mu t\ .\ Leaf : \alpha_{\obar}\ \oplus\ Node : t_{\obar} \otimes t_{\bot})\ (\mu n\ .\ Zero : \mathbf{1}_{\bot}\ \oplus\ Succ : n_{\bot}))_{\obar}$$

7 Extending the Prototype to Haskell

As mentioned previously, our language is pretty close to the Haskell Core language as used in the Glasgow compiler as an intermediate language after desugaring and type-checking the input program. However, the Core language is richer the one here as it covers various phenomena present in Haskell programs.

First of all, Haskell is higher-order. As our method works only for first order languages we are unable to do more than discover simple strictness when we encounter a higher-order function. That is, we produce poor but safe results for higher-order functions. A similar solution applies to non-uniformly recursive types. The only contexts we use over such types are $Ide_{\obar}$, $Bot_{\obar}$, $Bot_{\bot}$, $Ide_{\bot}$.

Another difference between the Core language and ours is that the former admits local definitions in the form of `let` and `letrec` expressions. This means that in the minimal function graphs we should not only keep the values for the functions defined at the entire level but also the values of functions defined by `let` and `letrec`s.

Similarly, Haskell allows partial function applications. This, and the presence of `let`s, `letrec`s and lambda expressions forces us to change the resulting type of the projection transformer. The transformer has to distinguish between the free and bound variables. So, when applied to an expression and a context the transformer returns a pair consisting of a tuple of safe contexts for the bound variables and a projection environment for the free variables. Such pairs will also be recorded in the minimal function graph. Only for the globally defined functions can we be sure the free-variables part of the result is empty.

The most serious problem comes from modules. If the strictness analyser is to return results, strictness properties have to be passed from one module to another. Currently, it is far from clear how to do this.

The only problem that type classes (the major innovation of Haskell) introduce is that many previously first order functions become higher-order when the dictionary is passed as an extra parameter. Again, this is a topic for further work.

8 Acknowledgements

Our thanks to Simon Peyton Jones whose many comments were very useful in improving the paper. This work was funded by ESPRIT basic research action 3124 *Semantique*.

Bibliography

[1] L. Augustsson. *A Compiler for Lazy ML.* Proceedings of Lisp and Functional Programming Conference, Austin, Texas, 1984.

[2] K. Davis and P. Wadler, *Strictness Analysis in 4D*, Third Annual Glasgow Workshop on Functional Programming, Ullapool, *Workshops in Computing*, S-V, 1990.

[3] R.J.M. Hughes and J. Launchbury, *Projections for Polymorphic Strictness Analysis*, To appear in *Mathematical Structures in Computer Science*, C.U.P.

[4] R.J.M. Hughes, *Projections for Polymorphic Strictness Analysis*, In *Category Theory in Computer Science*, Manchester, 1989.

[5] N.D. Jones and A. Mycroft, *Data Flow Analysis of Applicative Programs Using Minimal Function Graphs*, Proc. of the Thirteenth ACM Symposium on Principles of Programmng Languages, St. Petersburg, Florida, pp. 296-306, 1986

[6] J. Launchbury, *Projections for Specialisation*, in Bjørner, Ershov and Jones (eds), *Partial Evaluation and Mixed Computation.* Proceedings IFIP TC2 Workshop, Denmark, Oct 1987. North-Holland, 1988.

[7] J. Launchbury, *Projection Factorisations in Partial Evaluation*, Ph.D. thesis, Glasgow University. *Distinguished Dissertations in Computer Science*, Vol 1, CUP, 1991.

[8] P. Wadler and R.J.M. Hughes, *Projections for Strictness Analysis*, In *Functional Programming and Computer Architecture*, Portland, USA, LNCS 274, 1987