Strictness Analysis in 4D

Kei Davis
Dept. of Computing Science
University of Glasgow
Glasgow G12 8QQ
United Kingdom

Abstract

Strictness analysis techniques can be classified along four different dimensions: first-order vs. higher-order, flat vs. non-flat, low fidelity vs. high fidelity, and forward vs. backward. Plotting a table of the positions of known techniques within this space reveals that certain regions are densely occupied while others are empty. In particular, techniques for high-fidelity forward and low-fidelity backward analysis are well known, while those for low-fidelity forward and high-fidelity backward analysis are lacking. This paper fills in the gaps: the low-fidelity forward methods provide faster analyses than the high-fidelity forward methods, at the cost of accuracy, while the high-fidelity backward methods provide more information than the low-fidelity backward methods, at the cost of time.

1 Introduction

Strictness analysis is an important part of many compilers for lazy functional languages, and a wide variety of strictness analysis techniques have been proposed. It is not clear how all of the various techniques are related; this paper is in part an attempt to organise some of these methods, by determining their positions in a space of four properties. The second goal of the paper is to give analysis techniques for the heretofore unrepresented points in this space. The properties we consider are as follows.

First-order vs. higher-order. Analysis techniques may be applicable only to first-order languages, or more generally to higher-order languages. The ability to analyse higher-order expressions is important because higher-order programming is an essential part of the functional style.

Flat vs. non-flat. A flat semantic domain (e.g. the domain of integers or the domain of booleans) may be usefully abstracted to the two-point domain, since there are only two possible degrees of definedness: completely defined or completely undefined. This abstraction is too coarse for lazy data structures, for which it may be useful to differentiate between various degrees of definedness of the top-level structure (e.g. the spine of a list), and independently, between degrees of definedness of its subcomponents (e.g. the elements of a list). Analyses using only flat abstract domains will be called flat, while those using deeper domains will be called non-flat.

Low fidelity vs. high fidelity. The terms *low fidelity* and *high fidelity* are used to indicate how analyses are done with respect to either the free variables of an expression, or the formal parameters of a function definition. A low-fidelity analysis is one in which a separate analysis is done with respect to each free variable or formal parameter, and the results combined. In a high-fidelity analysis, all of the free variables or formal parameters are considered simultaneously, allowing possible relationships between their values to be

considered. This means that a high-fidelity analysis can potentially detect *joint strictness* in two or more parameters, that is, that the result of the function may be defined if at least one of the arguments is defined, but is certainly undefined if all of the parameters are undefined. A low-fidelity analysis cannot detect joint strictness since it cannot consider two or more parameters being undefined simultaneously. (This should not be confused with the distinction between *independent* and *relational* analyses, with differ in the abstraction of products. All of the analyses presented here are independent in this sense.)

Forward vs. backward. The term *forward* is used to describe abstract interpretations in which the goal is to discover the definedness of an expression given the definedness of its free variables, while *backward* describes interpretations in which the required definedness of the free variables is to be determined given the required definedness of the entire expression. In operational terms, a backward analysis determines the demand (required degree of evaluation) of the free variables of an expression, given the demand on the entire expression.

<table>
<tr><th colspan="2" rowspan="2"></th><th colspan="2">Forward</th><th colspan="2">Backward</th></tr>
<tr><th>Flat</th><th>Non-flat</th><th>Flat</th><th>Non-flat</th></tr>
<tr><td rowspan="2">H.F.</td><td>F.O.</td><td>[Myc81]</td><td rowspan="2">[Wad87]</td><td colspan="2" rowspan="2">(this paper)</td></tr>
<tr><td>H.O.</td><td>[BHA85]</td></tr>
<tr><td rowspan="2">L.F.</td><td>F.O.</td><td colspan="2" rowspan="2">(this paper)</td><td colspan="2">[WH87]</td></tr>
<tr><td>H.O.</td><td colspan="2">[Hug87]</td></tr>
</table>

Figure 1: Four dimensions of strictness analysis

Historically, the development of strictness analysis started as first-order, flat, high fidelity, and forward [Myc81]. This theory was extended to higher-order by [BHA85]. This in turn was extended to non-flat abstractions of lazy data structures by [Wad87].

The development of backward strictness analysis was motivated in part by the need for a method that could detect, for example, *head strictness* on lists—that the head field of a cons cell is evaluated whenever the cons cell itself is evaluated. Hughes describes a first-order, low fidelity, non-flat backward analysis in [Hug85] based on a concept of *contexts.* The development in this paper derives from the more formal development in [WH87], in which the analysis is based on abstract domains of projections.

In first-order backward analysis, the abstract value of an expression of non-function type is a backward value, and the abstract value of a function definition or primitive function is a forward value—a function from backward values to backward values. Higher-order backward analysis is complicated by the necessity of abstract values having both forward and backward components. Consider an analysis of the call *length fs*, where *fs* is a list of functions. We expect *length* to take a backward value associated with *fs* and return a backward value, just as in the first-order case. However, in an analysis of *head fs x*, we expect a forward value to be extracted from the abstract value of *fs*, to be applied to the abstract value of x. Thus the abstract value of *fs* must have both forward and backward

components.

As we show later, low-fidelity analysis of higher-order functions, both forward and backward, can be expected to give such poor information as to be useless. Presumably for this reason, the higher-order backward method outlined in [Hug87] is high fidelity in its higher-order components, but low fidelity in its first-order components.

Figure 1 summarises the development described thus far, and shows clearly an association of the high-fidelity property with the forward property, and the low-fidelity property with the backward property. On the basis of these associations, Hughes argues in [Hug87] that backward analysis is inherently faster than forward analysis. However, by giving low-fidelity forward and high-fidelity backward analysis techniques, we show that these associations are just artefacts of the historical development of the techniques, so that properly Hughes' argument is that low-fidelity analysis is faster than low-fidelity analysis.

This diagram is imperfect: ideally it would exclude low-fidelity analysis of higher-order higher-order functions (since it does not give useful analyses), and identify [Hug87] as a hybrid of low-fidelity first-order and high-fidelity higher-order backward analysis.

There is yet another dimension along which strictness analysis techniques may be characterised: monomorphic vs. polymorphic. This paper considers only techniques for the analysis of monomorphic languages. Techniques for polymorphic languages are described in [Abr85], [Hug88], and [Hug89b].

The rest of this paper is organised as follows. Section 2 defines the languages to be analysed. Section 3 describes first-order and higher-order low-fidelity forward analysis. Section 4 introduces projections and projection transformers, and recounts the essential properties of first-order low-fidelity and high-fidelity backward analysis. Section 5 describes higher-order high-fidelity backward analysis. Section 6 concludes.

2 Languages

This section defines the languages to analysed. The languages are assumed to be monomorphically typed. The language constructs and data types are representative of those in 'real' lazy functional languages: flat data types *Int* and *Bool* with strict operators + and =; a non-flat data type *List* with non-strict constructors [] and :, a **case** expression for decomposition of lists, and the conditional **if**. (Note the symbol : is also used to indicate type.)

In the domain equations, $\oplus$ denotes coalesced sum, and $\times$ denotes standard product. The constructions are standard, see e.g. [Sto77]. The symbol **1** is used to denote the one-point domain; its single element is denoted by (). Subscript $\perp$ denotes domain lifting.

2.1 A First-Order Language

Abstract Syntax

$x \in Var$	variables
$f \in FVar$	function variables
$e \in Exp$	expressions
$k \in Con$	numerals and boolean literals
$d \in Defs$	definition sets

$$
\begin{array}{lll}
e ::= & x & \text{variable} \\
| & k & \text{constant} \\
| & f\ e_1\ \ldots\ e_n & \text{function application} \\
| & e_1 + e_2 & \text{sum} \\
| & e_1 = e_2 & \text{equality} \\
| & \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 & \text{conditional} \\
| & [] & \text{empty list} \\
| & e_1 : e_2 & \text{construct list} \\
| & \textbf{case } e_0 \textbf{ of } \{[] \rightarrow e_1;\ (x : xs) \rightarrow e_2\} & \text{list decomposition}
\end{array}
$$

$$
d ::= \quad \{f_i\ x_{i,1}\ \ldots\ x_{i,n_i} = e_i \mid 1 \leq i \leq m\} \qquad \text{definition set}
$$

Semantic Domains

$$
\begin{array}{lll}
\mathit{Bool} & = \{\mathit{true},\ \mathit{false}\}_\perp & \text{booleans} \\
\mathit{Int} & = \{\ldots,\ -1,\ 0,\ 1,\ \ldots\}_\perp & \text{integers} \\
\mathit{List} & = \mathbf{1}_\perp \oplus (\mathit{Val} \times \mathit{List})_\perp & \text{lists} \\
\mathit{Val} & = \mathit{Bool} \oplus \mathit{Int} \oplus \mathit{List} & \text{values} \\
\mathit{Fun} & = \bigcup_{n=0}^{\infty} (\mathit{Val}^n \rightarrow \mathit{Val}) & \text{first order functions} \\
\mathit{FEnv} & = \mathit{FVar} \rightarrow \mathit{Fun} & \text{function environment} \\
\mathit{VEnv} & = \mathit{Var} \rightarrow \mathit{Val} & \text{variable environment}
\end{array}
$$

Semantic Functions

$$
\begin{array}{l}
\mathcal{E} : \mathit{Exp} \rightarrow \mathit{VEnv} \rightarrow \mathit{FEnv} \rightarrow \mathit{Val} \\
\mathcal{D} : \mathit{Dfns} \rightarrow \mathit{FEnv} \\
\mathcal{K} : \mathit{Con} \rightarrow \mathit{Val}
\end{array}
$$

The definitions of the semantic functions $\mathcal{E}$, $\mathcal{D}$, and $\mathcal{K}$ are standard, and are not repeated here. (See e.g. [Sto77].)

2.2 A Higher-Order Language

In the higher-order language, variables may have function values; given lambda abstractions, recursion equations are eliminated in favour of a fixed-point combinator.

Abstract Syntax

$$
\begin{array}{ll}
x \in \mathit{Var} & \text{variables} \\
e \in \mathit{Exp} & \text{expressions} \\
k \in \mathit{Con} & \text{numerals and boolean literals}
\end{array}
$$

$$
\begin{array}{lll}
e ::= & x & \text{variable} \\
| & k & \text{constant} \\
| & \lambda x.e & \text{lambda abstraction} \\
| & e_1\ e_2 & \text{function application} \\
| & \textbf{fix } e & \text{least fixed point} \\
| & e_1 + e_2 & \text{sum} \\
| & e_1 = e_2 & \text{equality} \\
| & \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 & \text{conditional} \\
| & [] & \text{empty list} \\
| & e_1 : e_2 & \text{construct list} \\
| & \textbf{case } e_0 \textbf{ of } \{[] \rightarrow e_1;\ (x : xs) \rightarrow e_2\} & \text{list decomposition}
\end{array}
$$

Semantic Domains

$$
\begin{array}{lll}
Bool = \{true,\ false\}_\perp & & \text{booleans} \\
Int = \{\ldots,\ -1,\ 0,\ 1,\ \ldots\}_\perp & & \text{integers} \\
Val = Bool \oplus Int \oplus List \oplus (Val \rightarrow Val) & & \text{values} \\
List = \mathbf{1}_\perp \oplus (Val \times List)_\perp & & \text{lists} \\
Env = Var \rightarrow Val & & \text{variable environment}
\end{array}
$$

Semantic Functions

$$
\begin{array}{l}
\mathcal{E} : Exp \rightarrow Env \rightarrow Val \\
\mathcal{K} : Con \rightarrow Val
\end{array}
$$

Again, the definitions of $\mathcal{E}$ and $\mathcal{K}$ are standard.

3 Forward Analysis

In the first-order forward analysis described in [Myc81], and the higher-order forward analysis described in [BHA85], the abstraction maps flat concrete domains to the two-point domain **2** with elements $\perp$ and $\top$, with $\perp \sqsubset \top$. Non-$\perp$ concrete values are mapped to $\top$ in **2**, and $\perp$ is mapped to $\perp$. The abstraction of function types is induced by the abstraction of of the base types, so that, for example, if concrete function f has type $(Int \rightarrow Int) \rightarrow Int$, then the abstraction $f^\#$ of f has type $(\mathbf{2} \rightarrow \mathbf{2}) \rightarrow \mathbf{2}$. Extension to non-flat domains is described in [Wad87]. For all of the analysis techniques, we will use $^\#$ to indicate the mapping of concrete domains to abstract domains, and of values in concrete domains to values in abstract domains. We give enough of the first-order abstract semantics to be able to contrast the high-fidelity and low-fidelity analysis techniques.

Abstract Semantic Domains

$$
\begin{array}{lll}
Val^\# = Bool^\# \oplus Int^\# \oplus List^\# & & \text{values} \\
List^\# = (Val^{\#}{}_\perp)_\perp & & \text{lists} \\
Fun^\# = \bigcup_{i=0}^{\infty} (Val^{\#^i} \rightarrow Val^\#) & & \text{first order functions} \\
FEnv^\# = FVar \rightarrow Val^\# & & \text{function environment} \\
VEnv^\# = Var \rightarrow Val^\# & & \text{variable environment}
\end{array}
$$

Abstract Semantic Functions

$$
\begin{array}{l}
\mathcal{E}^\# : Exp \rightarrow VEnv^\# \rightarrow FEnv^\# \rightarrow Val^\# \\
\mathcal{D}^\# : Defs \rightarrow FEnv^\#
\end{array}
$$

$$\mathcal{E}^\#[\![x]\!]\ \rho\ \sigma = \rho[\![x]\!]$$

$$\mathcal{E}^\#[\![k]\!]\ \rho\ \sigma = \top$$

$$\mathcal{E}^\#[\![f\ e_1\ \ldots\ e_n]\!]\ \rho\ \sigma = \sigma[\![f]\!]\ (\mathcal{E}^\#[\![e_1]\!]\ \rho\ \sigma)\ \ldots\ (\mathcal{E}^\#[\![e_n]\!]\ \rho\ \sigma)$$

$$
\begin{array}{rcl}
\mathcal{E}^\#[\![e_1 + e_2]\!]\ \rho\ \sigma & = & v_1 \sqcap v_2 \\
& & \text{where} \\
& & \quad v_1 = \mathcal{E}^\#[\![e_1]\!]\ \rho\ \sigma \\
& & \quad v_2 = \mathcal{E}^\#[\![e_2]\!]\ \rho\ \sigma
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E}^\#[\![e_1 = e_2]\!]\ \rho\ \sigma & = & v_1 \sqcap v_2 \\
& & \text{where} \\
& & \quad v_1 = \mathcal{E}^\#[\![e_1]\!]\ \rho\ \sigma \\
& & \quad v_2 = \mathcal{E}^\#[\![e_2]\!]\ \rho\ \sigma
\end{array}
$$

$$\mathcal{E}^{\#}[\![\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2]\!]\ \rho\ \sigma\ =\ \begin{array}{l} v_1 \sqcup v_2, \text{ if } v_0 = \top \\ \bot, \qquad \text{if } v_0 = \bot \\ \text{where} \\ \quad v_0 = \mathcal{E}^{\#}[\![e_0]\!]\ \rho\ \sigma \\ \quad v_1 = \mathcal{E}^{\#}[\![e_1]\!]\ \rho\ \sigma \\ \quad v_2 = \mathcal{E}^{\#}[\![e_2]\!]\ \rho\ \sigma \end{array}$$

$$\begin{array}{l} \mathcal{D}^{\#}[\![\{f_i\ x_{i,1}\ \ldots\ x_{i,n_i} = e_i \mid 1 \le i \le m\}]\!] \\ \quad = \mathit{fix}\ (\lambda\sigma.[(\lambda y_1 \ldots y_{n_i}.\mathcal{E}^{\#}[\![e_i]\!]\ [y_j/x_{i,j} \mid 1 \le j \le n_i]\ \sigma)/f_i \mid 1 \le i \le m]) \end{array}$$

A prototypical example is:

$$\mathit{cond}\ x\ y\ z = \textbf{ if } x \textbf{ then } y \textbf{ else } z.$$

Then

$$\mathit{cond}^{\#}\ \top\ y\ z = y \sqcup z,$$

$$\mathit{cond}^{\#}\ \bot\ y\ z = \bot.$$

This analysis detects that *cond* is strict in its first argument and jointly strict in its second and third arguments. We call these analysis techniques *high fidelity* since, in general, every possible combination of abstract arguments must be considered to fully determine the value of the abstract function.

3.1 First-Order Low-Fidelity Analysis

In first-order low-fidelity analysis, strictness in each of a function's arguments is determined individually. Each function definition $f\ x_1\ \ldots\ x_n = e$ gives rise to n abstract functions $f^{(1)}, \ldots, f^{(n)}$ of a single argument, corresponding to the abstraction with respect to each of the arguments x_j. Then $f^{(j)}\ \bot = \bot$ implies that f is strict in its j^{th} argument. The abstraction $f^{\#}$ of f is then taken to be

$$f^{\#} = \lambda x_1 \ldots \lambda x_n.f^{(1)}\ x_1 \sqcap \ldots \sqcap f^{(n)}\ x_n.$$

The abstract semantic domains and equations for the low-fidelity analysis are the same as for the high-fidelity analysis, except for the definition of $\mathcal{D}^{\#}$, which becomes

$$\begin{array}{l} \mathcal{D}^{\#}[\![\{f_i\ x_{i,1}\ \ldots\ x_{i,n_i} = e_i \mid 1 \le i \le m\}]\!] \\ \quad = \mathit{fix}\ (\lambda\sigma.[(\lambda y_1 \ldots \lambda y_{n_i}.\sqcap_{j=1}^{n_i} \mathcal{E}^{\#}[\![e_i]\!]\ \rho_{i,j}\ \sigma)/f_i \mid 1 \le i \le m]), \end{array}$$

where $\rho_{i,j}$ is shorthand for

$$[\top/x_{i,k} \mid 1 \le k \le n_i,\ k \ne j][y_j/x_{i,j}].$$

This definition does not explicitly define each $f_i^{(j)}$, $1 \le j \le n_i$, $1 \le i \le m$; they are implicitly defined by

$$f_i^{(j)} = \lambda x.f_i^{\#}\ \top \ldots \top\ x\ \top \ldots \top,$$

where x appears in the j^{th} argument position.

The advantage of the low-fidelity analysis is that for a function f of n arguments from abstract domains containing a_i elements each, $1 \le i \le n$, the abstraction of f is completely determined by only $\sum_{i=1}^{n} a_i$ combinations of argument values rather than $\prod_{i=1}^{n} a_i$ combinations for the high-fidelity analysis. The disadvantage is that joint strictness in two or more

arguments cannot be detected. An example for which the low-fidelity analysis is as good as the high-fidelity analysis is

$$\begin{aligned} \mathit{countdown}\ x\ y\ &=\ \textbf{if}\ y = 0\ \textbf{then}\ x\ \textbf{else}\ \mathit{countdown}\ x\ (y-1), \\ \mathit{countdown}^{\#}\ x\ y\ &=\ \begin{array}{l}((\top\ \sqcap\ \top)\ \sqcap\ (x\ \sqcup\ \mathit{countdown}^{\#}\ x\ (\top\ \sqcap\ \top))) \\ \sqcap\ ((y\ \sqcap\ \top)\ \sqcap\ (\top\ \sqcup\ \mathit{countdown}^{\#}\ \top\ (y\ \sqcap\ \top)))\end{array} \\ &=\ x\ \sqcap\ y, \end{aligned}$$

so low-fidelity analysis detects that *countdown* is strict in both of its arguments. However, for *cond* we have

$$\begin{aligned} \mathit{cond}^{\#}\ x\ y\ z\ &=\ \begin{array}{l}(x\ \sqcap\ (\top\ \sqcup\ \top)) \\ \sqcap\ (\top\ \sqcap\ (y\ \sqcup\ \top)) \\ \sqcap\ (\top\ \sqcap\ (\top\ \sqcup\ z))\end{array} \\ &=\ x, \end{aligned}$$

so strictness in the first argument is still detected, but joint strictness in the second and third is not. It is not hard to show that the the high-fidelity analysis always gives results as good or better than the low-fidelity analysis.

3.2 Higher-Order Low-Fidelity Analysis

Low-fidelity analysis of higher-order functions can be expected to give very poor results. Consider *apply* as the simplest example of a function that takes as arguments both a function and a value to which that function is applied:

$$\mathit{apply} = \lambda f.\lambda x.f\ x.$$

The high-fidelity analysis gives

$$\mathit{apply}^{\#}\ f\ x\ =\ f\ x.$$

The low-fidelity analysis gives

$$\begin{aligned} \mathit{apply}^{\#}\ f\ x &= f\ \top\ \sqcap\ \top\ x \\ &= f\ \top\ \sqcap\ \top \\ &= f\ \top. \end{aligned}$$

All information about the second argument is lost. The same problem causes the analyses of functions such as *map* and *fold* to yield similarly poor results. For this reason, we expect higher-order low-fidelity analysis to generally give very poor information.

4 First-Order Backward Analysis

4.1 Projections

We review the approach of [WH87].

A domain *projection* is a continuous idempotent function that approximates the identity function. In this paper, γ and δ will always denote projections, and α and β will always denote projection-valued variables. Projections form a complete lattice under the domain ordering $\sqsubseteq$, with the identity function *ID* as the top element, and the function *ABS* that maps every element to $\perp$ as the bottom element. A technical detail: the greatest lower

bound $\gamma \sqcap \delta$ of projections γ and δ is defined to be the greatest projection less than both γ and δ, since the greatest such function will not necessarily be a projection; for least upper bound, the function and projection coincide. The projections FST and SND are defined on pairs:

$$\begin{aligned} FST\ (u,v) &= (u,\bot), \\ SND\ (u,v) &= (\bot,v), \end{aligned}$$

for all values u and v. Then

$$\begin{aligned} FST \sqcup SND &= ID, \\ (FST \sqcap SND)\ (u,v) &= (\bot,\bot). \end{aligned}$$

Projections are used here to specify a degree of sufficient definedness of their arguments, by regarding those parts of their arguments which are mapped to $\bot$ as definitely not needed, and those parts left unchanged as possibly needed. For example, suppose that f is a function from pairs to pairs, and that for some particular application of f only the the first component of the result is needed. Then that instance of f may be safely replaced by $FST \circ f$. The terms *context* and *projection* are used interchangably; here f is said to be evaluated in context FST.

The particular context in which a function is evaluated may allow its argument to be less defined than in the general case. If

$$f\ (u,v) = (v,u),$$

then for f in context FST it is safe to apply SND to its argument, that is,

$$FST \circ f = FST \circ f \circ SND.$$

It is easy to show that for all projections γ and δ and functions f we have

$$\gamma \circ f = \gamma \circ f \circ \delta \text{ if and only if } \gamma \circ f \sqsubseteq f \circ \delta.$$

The projection δ is is not uniquely determined by f and γ, since for any f and γ, $\gamma \circ f \sqsubseteq f \circ ID$.

So far we have show how projections may be used to specify what part of a function's argument is not needed. Additionally, we wish to encode *necessity*: that some part of the argument is definitely required, or must be defined. To specify necessity with projections, all domains are extended by lifting. The new bottom-most element of each domain will be called $↳$, with domain D lifted written $D_{↳}$. We will interpret $\gamma\ u = ↳$ to mean that γ *requires* a value more defined than u. Every function $f : D_1 \to D_2$ is extended to a function in $D_{1↳} \to D_{2↳}$ by making it strict in $↳$, that is, $f\ ↳ = ↳$. The semantic function $\mathcal{E}$ is extended appropriately, with the additional condition that if its variable environment argument maps any variable to $↳$, then the result is $↳$. This ensures that functions such as $f\ x = 1$ with an argument that does not appear on the right-hand side of its definition are strict in $↳$. Thus the semantic domains become

$$\begin{aligned} Val &= Bool_{↳} \oplus Int_{↳} \oplus List_{↳} && \text{values} \\ Fun &= \bigcup_{n=0}^{\infty} (Val^n \to Val) && \text{first order functions} \end{aligned}$$

Let the projection STR be defined by

$$\begin{aligned} STR\ ↳ &= ↳, \\ STR\ u &= u,\ u \neq ↳. \end{aligned}$$

Simple strictness may be defined using STR, since

$$STR \circ f \sqsubseteq f \circ STR \text{ if and only if } f \text{ is strict.}$$

Every projection must map $\lightning$ to $\lightning$, so the definition of ABS in the lifted domains is

$$\begin{aligned} ABS\ \lightning &= \lightning, \\ ABS\ u &= \bot,\ u \neq \lightning. \end{aligned}$$

The projection ID is similar extended. If a function f makes no use of its argument, then for all γ,

$$\gamma \circ f \sqsubseteq f \circ ABS.$$

We also have

$$\gamma \circ f \sqsubseteq f \circ \delta \text{ implies } (\gamma \sqcup ABS) \circ f \sqsubseteq f \circ (\delta \sqcup ABS).$$

The least projection $FAIL$ is defined by

$$FAIL\ u = \lightning, \text{ for all } u.$$

Then for all f, $FAIL \circ f \sqsubseteq f \circ FAIL$.

The discussion so far generalises to functions of several arguments, but using function composition to indicate the context of a function's argument, e.g. $f \circ \delta$, is not adequate.

If a projection maps $\bot$ to $\lightning$, it is said to be *lift strict.* Thus STR and $FAIL$ are lift strict, but ABS and ID are not. In fact, STR is the greatest lift-strict projection, ABS is the least projection that is not lift strict. Since $STR \sqcup ABS = ID$ and $STR \sqcap ABS = FAIL$, the projections ID, STR, ABS, and $FAIL$ form a lattice; this lattice of projections will be denoted by the symbol $\diamond$.

It is useful to think of the context of an expression as defining a *demand* on the expression, that is, specifying a degree of evaluation of the expression which is certain to be performed. If the context maps some component of the value of an expression to $\bot$ for every value of that component, then the evaluation required to generate that component need not be performed. If the context maps some value to $\lightning$, then evaluation is certain to be performed at least far enough to guarantee that the result is not that value. Thus context ABS requires no evaluation. Context STR requires evaluation far enough to guarantee that the result is not $\bot$; for normal-order reduction, this means evaluation to WHNF. (For an expression of function type, it is not possible to determine that the expression does not denote the function $\bot$. Evaluation to WHNF is a safe degree of evaluation in context STR.) Context ID gives no information about how much evaluation will be performed. Context $FAIL$ indicates that no degree of evaluation yields an acceptable value.

Other useful projections are those that require constructor values. For lists, define for all γ and δ

$$\begin{aligned} NIL\ \lightning &= \lightning, & CONS\ \gamma\ \delta\ \lightning &= \lightning, \\ NIL\ \bot &= \lightning, & CONS\ \gamma\ \delta\ \bot &= \lightning, \\ NIL\ nil &= nil, & CONS\ \gamma\ \delta\ nil &= \lightning, \\ NIL\ (u:v) &= \lightning, & CONS\ \gamma\ \delta\ (u:v) &= (\gamma\ u) : (\delta\ v). \end{aligned}$$

Then, for example, the projection $CONS\ STR\ NIL$ requires its argument to be a list containing exactly one element, and that that element not be $\bot$. Analogous projections may be defined for any sum-of-products type.

4.2 Projection Transformers

A unary function from projections to projections will be called a *projection transformer* (PT). For the remainder of this paper, τ will always denote a projection transformer. Given a function f of n arguments, we wish to determine for each argument $x_i,\ 1 \leq i \leq n$, a corresponding PT τ_i such that for all γ,

$$\gamma\ (f\ x_1\ \ldots\ x_n)\ \sqsubseteq\ f\ x_1\ \ldots\ x_{i-1}\ (\tau_i\ \gamma\ x_i)\ x_{i+1}\ \ldots\ x_n.$$

If this inequality holds for each $\tau_i,\ 1 \leq i \leq n$, then

$$\gamma\ (f\ x_1\ \ldots\ x_n)\ \sqsubseteq\ f\ (\tau_1\ \gamma\ x_1)\ \ldots\ (\tau_n\ \gamma\ x_n)$$

also holds.

More generally, given an expression e, we will say that τ *is a safe abstraction of* e *with respect to* x *at* σ if for all projections γ and variable environments ρ,

$$\gamma\ (\mathcal{E}[\![e]\!]\ \rho\ \sigma)\ \sqsubseteq\ \mathcal{E}[\![e]\!]\ \rho[(\tau\ \gamma\ (\rho[\![x]\!]))/x]\ \sigma.$$

Thus the analysis of an expression is with respect to a particular free variable. We will usually assume that σ is implicit, and omit the qualification "at σ". If this condition holds for all lift-strict γ not equal to *FAIL*, then it will hold for all γ if

$$\begin{aligned} \tau\ \mathit{FAIL} &= \mathit{FAIL}, \\ \tau\ (\mathit{ABS}\ \sqcup\ \gamma) &= \mathit{ABS}\ \sqcup\ \tau\ \gamma,\ \gamma \text{ lift-strict}. \end{aligned}$$

Projection transformers satisfying these two equations will be said to have the *guard property*. We are only interested in PTs with the guard property, so PTs will be explicitly defined for arguments that are lift-strict and not equal to *FAIL*, with the understanding they satisfy these two equations. To indicate that PTs defined by lambda expressions have the guard property, the symbol $\bar{\lambda}$ is used instead of λ. So, for example

$$(\bar{\lambda}\alpha.\mathit{ABS})\ \mathit{FAIL}\ =\ \mathit{FAIL}.$$

Following are some useful facts about safety.

Fact 1. The least safe abstraction of x w.r.t. x is $\bar{\lambda}\alpha.\alpha$. The formal verification is trivial.

Fact 2. A safe abstraction of y w.r.t. x is $\bar{\lambda}\alpha.\mathit{ABS}$. We may think of this as formalising the statement that in evaluating the expression y in any environment ρ, modifying ρ to map x to $\perp$ does not change the result. Note that $\bar{\lambda}\alpha.\mathit{ABS}$ is the least safe abstraction of y w.r.t. x, since if an environment ρ maps x to $↯$, then $\mathcal{E}[\![y]\!]\ \rho\ \sigma\ =\ ↯$.

Fact 3. A safe abstraction of $x + y$ w.r.t. x is $\bar{\lambda}\alpha.\mathit{STR}$. Informally, this states that if $x + y$ must not be $\perp$, then x must not be $\perp$. In other words $x + y$ is strict in x.

Fact 4. The PT $\bar{\lambda}\alpha.\mathit{ID}$ is a safe abstraction of every expression w.r.t. every free variable. Note that $\bar{\lambda}\alpha.\mathit{ID}$ is the greatest PT with the guard property.

We will need PTs that map projections on constructor values to projections on their components. For lists we require PTs *HEAD* and *TAIL* such that for all u, v, and γ,

$$\gamma\ (u : v)\ \sqsubseteq\ (\mathit{HEAD}\ \gamma\ u) : (\mathit{TAIL}\ \gamma\ v).$$

The following definitions satisfy this requirement.

$$(\mathit{HEAD}\ \gamma)\ u\ =\ \bigsqcup_{v \in \mathit{List}\ D}\ \mathit{head}\ (\gamma\ (u : v)),$$

$$(\mathit{TAIL}\ \gamma)\ v\ =\ \bigsqcup_{u \in D}\ \mathit{tail}\ (\gamma\ (u : v)).$$

Then

$$HEAD\ (CONS\ \gamma\ \delta)\ =\ \gamma,$$

$$TAIL\ (CONS\ \gamma\ \delta)\ =\ \delta.$$

It will be convenient to define PTs by 'pattern-matching' lambda expressions, where the pattern is a projection constructor with variables in the argument positions. For lists we have

$$\bar{\lambda}NIL.e\ =\ \bar{\lambda}\alpha.\begin{cases} FAIL & \text{if } \alpha\ nil\ =\ \lightning, \\ e & \text{otherwise.} \end{cases}$$

$$\bar{\lambda}(CONS\ x\ y).f(x,y)\ =\ \bar{\lambda}\alpha.\begin{cases} FAIL, \text{ if } \bigsqcup_{u\in D}\bigsqcup_{v\in List\ D}\ \alpha\ (u:v)\ =\ \lightning, \\ f(HEAD\ \alpha,\ TAIL\ \alpha), \text{ otherwise.} \end{cases}$$

Intuitively, a PT defined in this way 'requires' that its argument accept some value or values, by returning *FAIL* if it does not. For example, the PT defined by $\bar{\lambda}(CONS\ x\ y).f(x,y)$ yields *FAIL* if its argument does not accept a cons node. This generalises to general sums-of-products in the obvious way.

The operators $\sqcup$ and $\sqcap$ are defined on PTs as follows. For all PTs τ_1 and τ_2,

$$\tau_1\ \sqcup\ \tau_2 =\ \bar{\lambda}\alpha.\tau_1\ \alpha\ \sqcup\ \tau_2\ \alpha,$$

$$\tau_1\ \sqcap\ \tau_2 =\ \bar{\lambda}\alpha.\tau_1\ \alpha\ \sqcap\ \tau_2\ \alpha.$$

Then PTs with the guard property form a lattice under the domain ordering $\sqsubseteq$.

Let e be an expression in which there are two instances of the free variable x. We will distinguish these two instances by substituting one instance of x by a new variable y, and call the resulting expression e'. Now suppose that for e' in context γ, we may safely apply projection δ_1 to x and projection δ_2 to y, that is, for all ρ,

$$\alpha\ (\mathcal{E}[\![e']\!]\ \rho\ \sigma)\ \sqsubseteq\ \mathcal{E}[\![e']\!]\ \rho[\delta_1\ (\rho[\![x]\!])/x]\ \sigma,$$

$$\alpha\ (\mathcal{E}[\![e']\!]\ \rho\ \sigma)\ \sqsubseteq\ \mathcal{E}[\![e']\!]\ \rho[\delta_2\ (\rho[\![y]\!])/y]\ \sigma.$$

In other words, for e in context α, δ_1 may be safely applied to one instance of x, and δ_2 to the other. Then a projection that may be safely applied to both instances of x is $\delta_1\ \&\ \delta_2$, where for all γ and δ,

$$\begin{array}{ll} (\gamma\ \&\ \delta)\ u\ =\ \lightning & \text{if } \gamma\ u\ =\ \lightning \text{ or } \delta\ u\ =\ \lightning, \\ (\gamma\ \&\ \delta)\ u\ =\ (\gamma\ \sqcup\ \delta)\ u & \text{otherwise.} \end{array}$$

The operator & is idempotent, commutative, associative, has *ABS* as identity, and distributes over $\sqcup$. Also, $\gamma\ \&\ FAIL\ =\ FAIL$ for all γ, and for all γ, δ, γ', δ',

$$(CONS\ \gamma\ \delta)\ \&\ (CONS\ \gamma'\ \delta')\ =\ CONS\ (\gamma\ \&\ \gamma')\ (\delta\ \&\ \delta').$$

The idea may be extended to projection transformers. Let e and e' be as above. Then if τ_1 is a safe abstraction of e' w.r.t. x, and τ_2 is is a safe abstraction of e' w.r.t. y, then $\tau_1\ \&\ \tau_2$ is a safe abstraction of e w.r.t. x, where

$$\tau_1\ \&\ \tau_2 =\ \bar{\lambda}\alpha.\tau_1\ \alpha\ \&\ \tau_2\ \alpha.$$

4.3 Low-Fidelity Analysis

The low-fidelity first-order backward analysis is described in [WH87]. We briefly recount its essential properties. Given an expression e and free variable x, the abstract semantics yields a PT τ that is a safe abstraction of e w.r.t. x. The abstraction of a function definition $f\ x_1\ \ldots\ x_n\ =\ e$ is a collection of functions $f^{(1)}, \ldots, f^{(n)}$, each of a single argument. Each of these functions maps a PT to a PT, such that for each $i,\ 1 \leq i \leq n$, and all ρ and γ,

$$\gamma\ (\mathcal{E}[\![f\ x_1\ \ldots\ x_n]\!]\ \rho\ \sigma)\ \sqsubseteq\ \mathcal{E}[\![f\ x_1\ \ldots\ x_n]\!]\ \rho[f^{(i)}\ \gamma\ (\rho[\![x]\!])/x]\ \sigma.$$

For *cond* the analysis gives

$$\begin{aligned} cond^{(1)} &= \bar{\lambda}\alpha.STR \\ cond^{(2)} &= \bar{\lambda}\alpha.ID \\ cond^{(3)} &= \bar{\lambda}\alpha.ID \end{aligned}$$

This reveals that *cond* is strict in its first argument, but tells nothing about strictness in its second and third arguments.

In general, the low-fidelity backward analysis, like the low-fidelity forward analysis, fails to detect joint strictness, and for a function of n arguments, requires only $\sum_{i=1}^{n} a_i$ combinations of abstract argument values to fully determine the abstraction of the function, where a_i is the size of the i^{th} abstract argument domain.

4.4 High-Fidelity Analysis

The high-fidelity first-order backward analysis is described in [DW90]. Like low-fidelity analysis, the high-fidelity analysis maps expressions to PTs. The abstraction of a function, however, is a function from PTs to PTs. Let x be a free variable, f a function of n arguments, and τ_i a safe abstraction of expression e_i w.r.t. x, $1 \leq i \leq n$. Then the abstraction $f^{\#}$ of f has the property that $f^{\#}\ \tau_1\ \ldots\ \tau_n$ is a safe abstraction of $f\ e_1\ \ldots\ e_n$ w.r.t. x. For example,

$$cond^{\#}\ \tau_1\ \tau_2\ \tau_3\ =\ \bar{\lambda}\alpha.\tau_1\ STR\ \&\ (\tau_2\ \alpha\ \sqcup\ \tau_3\ \alpha).$$

To analyse *cond* x y z w.r.t. x, take $\bar{\lambda}\alpha.\alpha$ as a safe abstraction of x w.r.t. x, and $\bar{\lambda}\alpha.ABS$ as safe abstractions of y and z w.r.t. x. Then

$$cond^{\#}\ (\bar{\lambda}\alpha.\alpha)\ (\bar{\lambda}\alpha.ABS)\ (\bar{\lambda}\alpha.ABS)\ =\ \bar{\lambda}.STR,$$

so *cond* x y z is strict in x. To analyse *cond* x y y w.r.t. y, take $\bar{\lambda}\alpha.ABS$ as a safe abstraction of x w.r.t. y, and $\bar{\lambda}\alpha.\alpha$ as a safe abstraction of y w.r.t. y. Then

$$cond^{\#}\ (\bar{\lambda}\alpha.ABS)\ (\bar{\lambda}\alpha.\alpha)\ (\bar{\lambda}\alpha.\alpha)\ =\ \bar{\lambda}\alpha.\alpha,$$

so *cond* x y y is strict in y, that is, *cond* is jointly strict in its second and third arguments.

In general, like high-fidelity forward analysis, the high-fidelity backward analysis can detect joint strictness in function arguments, and for a function of n arguments, requires $\prod_{i=1}^{n} a_i$ combinations of abstract argument values to fully determine the abstraction of the function, where a_i is the size of the i^{th} abstract argument domain.

5 Higher-Order Backward Analysis

In the first-order analysis, the abstract value of an expression e relative to a given free variable x is a PT mapping a projection, the context of e, to a projection that may be safely applied to every instance of x in e. We will call this PT the *backward abstraction* of e with

respect to x. In the first-order language, function definitions contain no free variables, so the values of functions and values of variables in an expression are completely independent. Given a function application, the function can only require particular definedness of a variable via its arguments, in which the variable may appear. Hence the abstract value of a function is just a function from the abstract values of its arguments to the abstract value of the application. We call this the *forward abstraction* of the function.

In the higher-order language, function definitions may contain free variables, so the evaluation of a function application may make some demand on a free variable, independent of the arguments. It may also make some additional demand on a variable via its arguments, as in the first-order case. For example, evaluation of the expression

$$(\textbf{if } b \textbf{ then } f \textbf{ else } g)\ b$$

requires that b be defined, independent of f and g. Application of f or g to b may also make some demand on b. This suggests that the abstract value of a function should encode two components: a backward abstraction giving demand on a given variable as a function of the surrounding context, independent of its arguments, and a forward abstraction, as in the first-order case. This is close to what we want.

5.1 Abstract Semantic Domains

The abstract value of every expression is a pair, consisting of a backward abstraction and a forward abstraction. The backward abstraction, as in the first-order case, is a projection transformer.

In the first-order analysis, the concept of a PT being a safe backward abstraction of an expression with respect to particular instances of a free variable was introduced. For informally explaining the higher-order analysis we introduce an operational element. Given a reduction rule—here normal order—and a specified degree of evaluation, e.g. to WHNF, suppose that for expression e in context γ, we may safely apply δ to those instances of the free variable x *referenced* during the reduction process. For example, for

$$\textbf{if } b \textbf{ then } (\lambda x.e_1) \textbf{ else } (\lambda x.e_2)$$

in context STR (indicating that the expression is certain to be evaluated to WHNF), we may safely apply STR to the instance of b immediately following **if**. We will not formalise the concept of 'reference during the reduction process'; it is for aiding intuition and will not be part of any formal definition.

We will say that τ is a safe backward abstraction of e with respect to x in reducing e to a given form if, for e in any context γ, we may safely replace those instances of x referenced during the reduction process by $\tau\ \gamma\ x$. For example, if no reduction of e occurs, then $\bar{\lambda}\alpha.ABS$ is a safe backward abstraction of e. In the first-order analysis, the backward abstraction of an expression corresponds to complete evaluation of the expression, that is, the resulting PT is a safe backward abstraction of the expression with respect to every instance of the relevant variable.

If τ is a safe backward abstraction of e in reducing e by some amount, then τ is also safe for any lesser amount of reduction. For the higher-order analysis we assume that expressions are reduced as far as possible, except for expressions or subexpressions of function type, which are reduced to WHNF. The resulting PT will then be safe for any lesser degree of reduction.

Let expression e have type T, and let $*$ be the type of free variable x. Then the type of any backward abstraction of e with respect to x is $|T| \rightarrow |*|$, where $|T|$ is the type of projections over T, and similarly for $|*|$.

Let A map the type of an expression to the type of its abstract value, and F map the type of an expression to the type of its forward abstraction. We have already decided that

$$A\ T\ =\ (|T|\ \rightarrow\ |*|)\ \times\ F\ T.$$

Since we intend to determine strictness using projections, the forward abstraction of an expression of base type needn't even come from a domain as rich as **2**, so we will take it from the one-point domain **1**, with sole element (). Thus

$$F\ K\ =\ \mathbf{1}.$$

The forward abstraction of an expression of function type is a function from abstract values to abstract values, so

$$F\ (U\ \rightarrow\ V)\ =\ A\ U\ \rightarrow\ A\ V.$$

We could take the forward abstraction of an expression of list type to be a list of forward abstractions. However, so that the abstract domains for any given program are finite (assuming the projection domains to be finite), we will represent this list by the least upper bound of all of its elements, so

$$F\ (\mathit{List}\ T)\ =\ F\ T.$$

5.2 Safety

The criteria for an abstract value $(\tau,\ \kappa)$ being a *safe* abstraction of an expression e with respect to x depend on the type of e:

Case $e : K$**.** For all ρ and γ we have

$$\gamma\ (\mathcal{E}[\![e]\!]\ \rho)\ \sqsubseteq\ \mathcal{E}[\![e]\!]\ \rho[(\tau\ \gamma(\rho[\![x]\!]))/x].$$

This is the same requirement as for the first-order case.

Case $e : U\ \rightarrow\ V$**.** Before giving the formal condition for safety we informally discuss the abstract semantics for application.

Intuitively, we may think of τ as being a safe abstraction of e in reducing e to WHNF. Let e_0 be an expression with safe abstraction σ_0 with respect to x. In applying e to e_0, e is first evaluated to WHNF, so the projection transformer mapping the context of $e\ e_0$ to the context of x in e is $\bar{\lambda}\alpha.\tau\ \mathit{STR}$. The resulting expression, with forward abstraction κ, is then applied to e_0, which has abstract value σ_0, giving some result with abstract value $(\tau_1,\ \kappa_1)\ =\ \kappa\ \sigma_0$. This result is in the same context as the application $e\ e_0$, so the two backward abstractions are &-combined:

$$(\bar{\lambda}\alpha.\tau\ \mathit{STR})\ \&\ \tau_1\ =\ \bar{\lambda}\alpha.\tau\ \mathit{STR}\ \&\ \tau_1\ \alpha.$$

The forward abstraction of the result is then κ_1. Thus the rule for abstract application, which will be denoted by infix $\star$, is

$$\begin{aligned}\sigma_0\ \star\ \sigma_1\ =\ &(\bar{\lambda}\alpha.\tau_0\ \mathit{STR}\ \&\ \tau_2\ \alpha,\ \kappa_2)\\ &\text{where}\\ &\quad(\tau_0,\ \kappa_0)\ =\ \sigma_0\\ &\quad(\tau_2,\ \kappa_2)\ =\ \kappa_0\ \sigma_1.\end{aligned}$$

Formally, for all expressions e_0 with safe abstractions σ_0 with respect to x, the result of the abstract application $(\tau,\ \kappa)\ \star\ \sigma_0$ must be a safe abstraction of $(e\ e_0)$ with respect to x.

Case $e : List\ T$**.** There are two pairs of conditions. Firstly, for all ρ,

$$NIL\ (\mathcal{E}[\![e]\!]\ \rho)\ \sqsubseteq\ \mathcal{E}[\![e]\!]\ \rho[(\tau\ NIL\ (\rho[\![x]\!]))/x],$$

$$(CONS\ ABS\ ABS)\ (\mathcal{E}[\![e]\!]\ \rho)\ \sqsubseteq\ \mathcal{E}[\![e]\!]\ \rho[(\tau\ (CONS\ ABS\ ABS)\ (\rho[\![x]\!]))/x].$$

Secondly, $head^{\#}\ \star\ (\tau,\ \kappa)$ and $tail^{\#}\ \star\ (\tau,\ \kappa)$ must be safe abstractions of *head* e and *tail* e with respect to x, respectively, where

$$head^{\#}\ =\ (\bar{\lambda}\alpha.ABS,\ \lambda(\tau,\ \kappa).(\bar{\lambda}\alpha.\tau\ (CONS\ \alpha\ ABS),\ \kappa)),$$

$$tail^{\#}\ =\ (\bar{\lambda}\alpha.ABS,\ \lambda(\tau,\ \kappa).(\bar{\lambda}\alpha.\tau\ (CONS\ ABS\ \alpha),\ \kappa)).$$

Intuitively, the first pair of conditions requires that τ be a safe backward abstraction of e with respect to x in evaluating e to WHNF. Note that the first of these is trivially satisfied when the value of e is cons, and the second when the value of e is nil. The second pair of conditions requires that the subcomponents—the head and tail of the list—be safely abstracted. This pair of conditions is trivially satisfied when the value of e is nil.

The definition of safety is recursively defined, firstly on the structure of the types expressions, and secondarily, because of the clause involving tails of lists, on the lengths of lists. Except for the case of infinite lists, the recursion is clearly well-founded. We conjecture that even in the case of infinite lists, the safety condition is still meaningful.

5.3 Abstract Semantic Function

The abstract semantic function $\mathcal{E}^{\#}$ must satisfy the following safety condition. If e is an expression with free variables $z_i,\ 1 \le i \le n$, and e_i is an expression with safe abstraction σ_i with respect to $x,\ 1 \le i \le n$, then $\mathcal{E}^{\#}[\![e]\!]\ [\sigma_i/z_i \mid 1 \le i \le n]$ is a safe abstraction of $e[e_i/z_i \mid 1 \le i \le n]$ with respect to x.

Abstract Semantic Domains

Val	$= Bool_{\downarrow} \oplus Int_{\downarrow} \oplus List_{\downarrow} \oplus (Val \to Val)_{\downarrow}$	values
$Proj$	$\subseteq Val \to Val$	projections
BA	$= Proj \to Proj$	backward abstractions
FA	$= \mathbf{1}_{\perp} \oplus (Val^{\#} \to Val^{\#})$	forward abstractions
$Val^{\#}$	$= BA \times FA$	abstract values
$VEnv^{\#}$	$= Var \to Val^{\#}$	variable environment

Abstract Semantic Function. The type of the abstract semantic function is

$$\mathcal{E}^{\#} : Exp \to VEnv^{\#} \to Val^{\#}.$$

In the following explanation of the definition of $\mathcal{E}^{\#}$ we will implicitly use the fact that if τ is a safe backward abstraction of some expression e in reducing e to e', and τ' is a safe backward abstraction of e' in reducing e' to e'', then $\tau\ \&\ \tau'$ is a safe backward abstraction of e in reducing e to e''.

All constants are of base type, so the forward abstraction of a constant is $()$. The backward abstraction is the same as for the first-order analysis, $\bar{\lambda}\alpha.ABS$, so

$$\mathcal{E}^{\#}[\![k]\!]\ \rho\ =\ (\bar{\lambda}\alpha.ABS, ()).$$

It is very easy to show that this definition satisfies the safety condition.

The abstract value of a variable is just its value in the enclosing environment:

$$\mathcal{E}^{\#}[\![x]\!]\ \rho\ =\ \rho[\![x]\!].$$

This definition trivially satisfies the safety condition.

The rule for application is

$$\mathcal{E}^{\#}[\![e_0\ e_1]\!]\ \rho\ =\ (\mathcal{E}^{\#}[\![e_0]\!]\ \rho)\ \star\ (\mathcal{E}^{\#}[\![e_1]\!]\ \rho).$$

If e_0 and e_1 are safely abstracted, then the definition trivially satisfies the safety condition.

A lambda expression is already in WHNF, so its backward abstraction is $\bar{\lambda}\alpha.ABS$. Some work is required to justify the the forward abstraction; intuitively, it is just like the rule for the ordinary lambda-calculus.

$$\mathcal{E}^{\#}[\![\lambda x.e]\!]\ \rho\ =\ (\bar{\lambda}\alpha.ABS,\ \lambda\sigma.\mathcal{E}^{\#}[\![e]\!]\ \rho[\sigma/x])$$

The abstraction of **fix** e is the least fixed point of the abstraction of e. Abstract application must be made explicit.

$$\mathcal{E}^{\#}[\![\mathbf{fix}\ e]\!]\ \rho\ =\ \mathit{fix}\ (\lambda\sigma.(\mathcal{E}^{\#}[\![e]\!]\ \rho)\ \star\ \sigma)$$

Addition is defined on integers, and equality is defined on integers and booleans. Assuming that e_1 and e_2 are safely abstracted, it is very easy to show that the following definitions satisfy the safety condition. The rule for addition has already been informally discussed; the same discussion applies to the rule for equality.

$$\begin{aligned}\mathcal{E}^{\#}[\![e_1 + e_2]\!]\ \rho\ &= (\bar{\lambda}\alpha.\tau_1\ STR\ \&\ \tau_2\ STR,\ ())\\ &\quad \text{where}\\ &\quad\quad (\tau_1, \kappa_1)\ =\ \mathcal{E}^{\#}[\![e_1]\!]\ \rho\\ &\quad\quad (\tau_2, \kappa_2)\ =\ \mathcal{E}^{\#}[\![e_2]\!]\ \rho\end{aligned}$$

$$\begin{aligned}\mathcal{E}^{\#}[\![e_1 = e_2]\!]\ \rho\ &= (\bar{\lambda}\alpha.\tau_1\ STR\ \&\ \tau_2\ STR,\ ())\\ &\quad \text{where}\\ &\quad\quad (\tau_1, \kappa_1)\ =\ \mathcal{E}^{\#}[\![e_1]\!]\ \rho\\ &\quad\quad (\tau_2, \kappa_2)\ =\ \mathcal{E}^{\#}[\![e_2]\!]\ \rho\end{aligned}$$

In evaluating **if** e_0 **then** e_1 **else** e_2 in a lift-strict context, either e_0 is evaluated to WHNF, followed by evaluation of e_1 in the same context as the entire expression, giving backward abstraction $\bar{\lambda}\alpha.\tau_0\ STR\ \&\ \tau_1\ \alpha$, or, e_0 is evaluated to WHNF, followed by evaluation of e_2 in the same context as the entire expression, giving backward abstraction $\bar{\lambda}\alpha.\tau_0\ STR\ \&\ \tau_2\ \alpha$. Since either may occur, we take the least upper bound of these two backward abstractions as a safe for both alternatives. Similarly, assuming that κ_1 and κ_2 are safe forward abstractions of e_1 and e_2, respectively, then $\kappa_1\ \sqcup\ \kappa_2$ is a safe abstraction of both. Thus

$$\begin{aligned}\mathcal{E}^{\#}[\![\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]\!]\ \rho\ &= (\bar{\lambda}\alpha.\tau_0\ STR\ \&\ (\tau_1\ \alpha\ \sqcup\ \tau_2\ \alpha),\ \kappa_1\ \sqcup\ \kappa_2)\\ &\quad \text{where}\\ &\quad\quad (\tau_0, \kappa_0)\ =\ \mathcal{E}^{\#}[\![e_0]\!]\ \rho\\ &\quad\quad (\tau_1, \kappa_1)\ =\ \mathcal{E}^{\#}[\![e_1]\!]\ \rho\\ &\quad\quad (\tau_2, \kappa_2)\ =\ \mathcal{E}^{\#}[\![e_2]\!]\ \rho.\end{aligned}$$

Assuming that subexpressions e_0, e_1, and e_2 are safely abstracted, it is not hard to show that the definition satisfies the safety condition.

Evaluation of [] makes no demand on any variable, but the context must accept a nil value, so the backward abstraction is $\bar{\lambda}NIL.ABS$. The forward abstraction of a list is the

least upper bound of the forward abstractions of its elements; [] has no elements and $\bot$ is the identity of $\sqcup$, so we take $\bot$ as its forward abstraction, so

$$\mathcal{E}^{\#}[\![[]]\!]\ \rho\ =\ (\bar{\lambda}NIL.ABS,\ \bot).$$

This definition is easily shown to satisfy the safety condition.

Intuitively, for $e_1 : e_2$ in context $CONS\ \alpha\ \beta$, e_1 is in context α and e_2 is in context β, so the backward abstraction of $e_1 : e_2$ is $\bar{\lambda}(CONS\ \alpha\ \beta).\tau_1\ \alpha\ \&\ \tau_2\ \beta$. If κ_1 is a safe forward abstraction of e_1, and κ_2 is a safe forward abstraction of every element of e_2, then $\kappa_1\ \sqcup\ \kappa_2$ is a safe forward abstraction of every element of $e_1 : e_2$. Thus

$$\begin{aligned}\mathcal{E}^{\#}[\![e_1 : e_2]\!]\ \rho\ &= (\bar{\lambda}(CONS\ \alpha\ \beta).\tau_1\ \alpha\ \&\ \tau_2\ \beta,\ \kappa_1\ \sqcup\ \kappa_2)\\ &\quad \text{where}\\ &\qquad (\tau_1, \kappa_1)\ =\ \mathcal{E}^{\#}[\![e_1]\!]\ \rho\\ &\qquad (\tau_2, \kappa_2)\ =\ \mathcal{E}^{\#}[\![e_2]\!]\ \rho.\end{aligned}$$

It is not hard to show that if e_1 and e_2 are safely abstracted, then this definition satisfies the safety condition.

Assuming that the subexpressions are safely abstracted, it is not too hard to show that the definition for **case** satisfies the safety condition:

$$\begin{aligned}&\mathcal{E}^{\#}[\![\textbf{case}\ e_0\ \textbf{of}\ \{[]\ \rightarrow\ e_1;\ (x : xs)\ \rightarrow\ e_2\}]\!]\ \rho\\ &= (\bar{\lambda}\alpha.(\tau_0\ NIL\ \&\ \tau_1\ \alpha)\ \sqcup\ (\tau_0\ (CONS\ ABS\ ABS)\ \&\ \tau_2\ \alpha),\ \kappa_1\ \sqcup\ \kappa_2)\\ &\quad \text{where}\\ &\qquad (\tau_0, \kappa_0)\ =\ \mathcal{E}^{\#}[\![e_0]\!]\ \rho\\ &\qquad (\tau_1, \kappa_1)\ =\ \mathcal{E}^{\#}[\![e_1]\!]\ \rho\\ &\qquad (\tau_2, \kappa_2)\ =\ \mathcal{E}^{\#}[\![e_2]\!]\ \rho[(head^{\#}\ \star\ (\tau_0,\ \kappa_0))/x,\ (tail^{\#}\ \star\ (\tau_0,\ \kappa_0))/xs].\end{aligned}$$

Evaluation of the expression in a lift-strict context requires evaluation of e_0 to value nil followed by evaluation of e_1 in the same context as the entire expression, hence backward abstraction $\bar{\lambda}\alpha.\tau_0\ NIL\ \&\ \tau_1\ \alpha$, or, evaluation of e_0 is evaluated to a cons value followed by evaluation of e_2 in the same context as the entire expression, hence backward abstraction $\tau_0\ (CONS\ ABS\ ABS)\ \&\ \tau_2\ \alpha$. Since either of these may occur, we take the least upper bound as safely abstracting both. Similarly, we take as the resulting forward abstraction the least upper bound of the forward abstractions of the possible results, e_1 and e_2. The abstract environment for e_2 is the same as the environment of the entire expression, augmented by appropriate values for the newly introduced variables.

5.4 Example

Let

$$\begin{aligned}compose\ =\ \textbf{fix}\ (\lambda compose.\lambda fs.\lambda x.&\textbf{case}\ fs\ \textbf{of}\\ &[]\ \ \ \ \rightarrow\ x\\ &g : gs\ \rightarrow\ g\ (compose\ gs\ x)).\end{aligned}$$

Our goal is to find a safe backward abstraction of *compose fs x* with respect to *fs*, under the assumption that every element of *fs* is strict. Choose the backward abstraction τ_f of *fs* to be safe with respect to *fs*, that is, $\bar{\lambda}\alpha.\alpha$. Choose the forward abstraction κ_{fs} to be the greatest value such that each element of *fs* is strict, that is

$$\kappa_{fs}\ =\ \lambda(\tau,\ \kappa).(\bar{\lambda}\alpha.\tau\ STR,\ \top).$$

Choose the backward abstraction τ_x of x to be safe with respect to *fs*, i.e. $\bar{\lambda}\alpha.ABS$. The forward abstraction κ_x of x does not affect the result. Then the first component of

$$\mathcal{E}^{\#}[\![compose\ fs\ x]\!]\ [(\tau_{fs}, \kappa_{fs})/fs,\ (\tau_x, \kappa_x)/x]$$

is a safe backward abstraction *compose fs x* with respect to *fs*. The calculation is straightforward; the result is

$$\mu\tau.\bar{\lambda}\alpha.NIL\ \sqcup\ CONS\ STR\ (\tau\ STR)$$

$$= \bar{\lambda}\alpha.FIN\ STR$$

where for all α, $FIN\ \alpha\ =\ NIL\ \sqcup\ CONS\ \alpha\ (FIN\ \alpha)$. We conclude that if each element of *fs* is strict, then for the expression *compose fs x* in a strict context, it is safe to apply *FIN STR* to *fs*.

6 Conclusion

A new low-fidelity first-order forward analysis technique and high-fidelity higher-order backward analysis technique have been presented. Both appear to have potential for practical use. We have shown that low-fidelity higher-order forward analysis can be expected to give very poor results; we hypothesise similarly poor results from a low-fidelity higher-order backward analysis.

References

[Abr85] Abramsky, S. "Strictness analysis and polymorphic invariance." In *Proceedings of the Workshop on Programs a Data Objects* (Copenhagen). H. Ganzinger and N. Jones, eds. LNCS 217. Springer-Verlag, Berlin, 1985.

[AH87] Abramsky, S. and Hankin, C. (eds.). *Abstract Interpretation of Declarative Languages.* Ellis-Horwood, 1987.

[AH87b] Abramsky, S. and Hankin, C. "An introduction to abstract interpretation." Chapter 1 of [AH87b]

[Bur90] Burn, G.L. "A relationship between abstract interpretation and projection analysis." *POPL (San Francisco, January, 1990).*

[BHA85] Burn, G., Hankin, C., and Abramsky, S. "The theory of strictness analysis for higher-order functions." In *Proceedings of the Workshop on Programs a Data Objects* (Copenhagen). H. Ganzinger and N. Jones, eds. LNCS 217. Springer-Verlag, Berlin, 1985

[CP85] Clack, C. and Peyton Jones, S. "Strictness analysis—A practical approach." In *Proceedings of Functional Programming Languages and Computer Architecture* (Nancy, France). J.-P. Jounannaud, ed., LNCS 201. Springer-Verlag, Berlin, 1985.

[Dav89] Davis, K. "Trading accuracy for efficiency in forwards strictness analysis." Unpublished report, 1989.

[DW90] Davis, K. and Wadler, P. "Strictness analysis: Proved and improved." In *Functional Programming: Proceedings of the 1989 Glasgow Workshop, 21-23 August 1989, Fraserburgh Scotland.* K. Davis and J. Hughes, eds. Springer Workshops in Computing. Springer-Verlag, 1990.

[Hug85] "Strictness detection in non-flat domains." *Proceedings of the Workshop on Programs a Data Objects* (Copenhagen). H. Ganzinger and N. Jones, eds. LNCS 217. Springer-Verlag, Berlin, 1985

[Hug87] Hughes, R.J.M. *Backwards Analysis of Functional Programs.* Departmental Research Report CSC/87/R3, Department of Computing Science, University of Glasgow, 1987.

[Hug88] Hughes, R.J.M. "Abstract Interpretation of First-Order Polymorphic Functions." *Proceedings of the 1988 Glasgow Workshop on Functional Programming.* August 2-5, 1988, Rothesay, Isle of Bute, Scotland. Department of Computing Science, University of Glasgow, Glasgow, Scotland.

[Hug89b] Hughes, R.J.M. "Projections for polymorphic strictness analysis." In *Category Theory and Computer Science* (Manchester). D.H. Pitt, D.E. Rydeheard, P. Dybjer, A.M. Pitts, A. Poigne, eds. LNCS 389. Springer Verlag, Berlin, 1989.

[Myc81] Mycroft, A. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* Ph.D. thesis, University of Edinburgh, 1981.

[Sto77] Stoy, Joseph E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge, Massachusetts, 1977.

[Wad87] Wadler, P. "Strictness analysis on non-flat domains by abstract interpretation over finite domains." Chapter 12 of [AH87].

[WH87] Wadler, P., and Hughes, J. *Projections for Strictness Analysis.* Report 35, Programming Methodology Group, Department of Computer Sciences, Chalmers University of Technology and University of Göteborg, Göteborg, Sweden, 1987.