Projections for Strictness Analysis*

Philip Wadler
Programming Research Group, Oxford University
and Programming Methodology Group, Chalmers University, Göteborg

R.J.M. Hughes
Department of Computer Science, University of Glasgow

May 1987

Abstract

Contexts have been proposed as a means of performing strictness analysis on non-flat domains. Roughly speaking, a *context* describes how much a sub-expression will be evaluated by the surrounding program. This paper shows how contexts can be represented using the notion of *projection* from domain theory. This is clearer than the previous explanation of contexts in terms of continuations. In addition, this paper describes *finite domains* of contexts over the non-flat list domain. This means that recursive context equations can be solved using standard fixpoint techniques, instead of the algebraic manipulation previously used.

Praises of lazy functional languages have been widely sung, and so have some curses. One reason for praise is that laziness supports programming styles that are inconvenient or impossible otherwise [Joh87, Hug84, Wad85a]. One reason for cursing is that laziness hinders efficient implementation.

Still, acceptable efficiency for lazy languages is at last being achieved. This is done by means of graph reduction [Pey87], as found in the G-machine [Aug84, Joh84] and the Ponder implementation [FW86], among others. The essential trick is to evaluate an expression immediately, when this is safe, rather than to construct a graph. Strictness analysis can reveal more places where this optimisation is safe. In the Ponder implementation, strictness analysis speeds up some programs by a factor of two or more. In addition, strictness analysis may enable other optimisations, such as destructive updating of arrays [HB85].

Accordingly, strictness analysis has received much attention; see [AH87] for a collection of some recent work. An elegant approach to strictness analysis is abstract interpretation. This approach was first applied by Mycroft [Myc81], and later extended to higher-order

*Presented at the *Third International Conference on Functional Programming Languages and Computer Architecture*, Portland, Oregon, September 1987.

languages [BHA85, HY85] and polymorphism [Abr85]. For an excellent introduction, see [CP85].

A remaining question of great interest was how to perform strictness analysis for data types over non-flat domains, such as lazy lists. An early proposal in this direction was made by Hughes, based on analysis of the *context* in which an expression may be evaluated [Hug85, Hug87a]. The method could determine useful information about strictness in programs using lazy lists. But it had three drawbacks.

First, it was not clear exactly what a context was. The first paper was somewhat informal, and concluded "a proper treatment would be welcome". This was provided by the second paper, which modeled contexts as abstractions of sets of continuations. The model was less than completely intuitive, and the proofs involved were lengthy.

Second, context analysis yielded equations that were difficult to solve. The equations had to be solved by algebraic manipulation, and the descriptions of the manipulations required ran to many pages. Further, exact simplifications were not always possible, and heuristics were required to decide what approximations to introduce. An example in [Hug85] shows how an apparently reasonable heuristic can lead to an unreasonably bad approximation.

Third, the method applied only to first-order, untyped languages. Extensions to higher-order, polymorphic languages would need to be developed.

Meanwhile, Wadler, in part inspired by Hughes' work, discovered a different method of analysing strictness on non-flat domains, with none of the above drawbacks [Wad87]. First, the method was a straightforward extension of abstract interpretation, and so built on existing mathematical foundations and intuitions. Second, like other work on abstract interpretation, it used finite domains. Fixpoints could be found by straightforward techniques [CP85], and methods for finding fixpoints efficiently could be directly applied [PC87, YH86]. No algebraic manipulation was required. Third, since abstract interpretation had already been extended to include higher-order languages and polymorphism, so did this method.

An open and shut case? Not quite. As it turns out, context analysis reaches the places abstract interpretation cannot reach. As we shall see in the next section, there are primarily two kinds of strictness of interest for lazy lists, head strictness and tail strictness. Context analysis can find both. Abstract interpretation can find tail strictness, and it can find head strictness when it is combined with tail strictness. But it cannot find head strictness alone. That's a shame. A major paradigm in lazy functional programming involves functions that read a bit of the input list and then produce a bit of the output list, acting like a coroutine. Such functions are often head strict, but cannot be tail strict. So the additional power of context analysis is important.

This paper provides a new description of contexts that addresses previous shortcomings.

First, a simple, and we believe intuitive, explanation of contexts is given. The notion of context is identified with the notion of *projection* from domain theory. The proofs involved are simpler than those in [Hug87a].

Second, finite domains are given for contexts over lists. This means that the standard fixpoint methods can be used, and algebraic manipulation is no longer required.

Third ... well, two out of three isn't bad. The method is still limited to a first-order, monomorphic language. However, there are reasons to believe that context analysis will follow in the footsteps of abstract interpretation, and be extended to higher-order functions and polymorphism; one way of doing so is outlined in [Hug87b]. It is hoped that the new explanation of contexts given here will aid in this task.

The history of this paper is as follows. Wadler discovered how to represent contexts by projections (Sections 2–4, 6–8) after studying the work of Hughes [Hug87a]. Hughes discovered how to define finite domains for contexts over lists (Sections 5, 6.6) after studying the work of Wadler [Wad87]. The paper itself was written by Wadler.

Contexts have close relations to other work on strictness analysis, including that of Burn [Bur87], Dybjer [Dyb87], Hall [HW87], and Wray [Wra86, FW86]. Contexts may also be applied to Bjerner's work on analysing time and space complexity [Bje87], and to analysing pre-order traversal [Wad85b]. These issues are discussed in Section 8.

This paper assumes the reader knows some domain theory; for an introduction, see one of [Sco82, Sto77, Sch86]. The domains used are consistently complete, algebraic cpos. Familiarity with abstract interpretation or the authors' previous work is helpful but not required.

This paper is organized as follows. Section 1 introduces head strictness and tail strictness. Section 2 formalizes these with projections. Section 3 extends the method to deal with ordinary strictness. Section 4 discusses finite domains of contexts over flat domains. Section 5 discusses finite domains of contexts over lists. Section 6 develops the fundamentals of context analysis. Section 7 presents examples. Section 8 compares related work. Section 9 concludes.

1 Head strictness and tail strictness

As usual, let $:$ be the list construction operator, cons, so that $1 : 2 : []$ denotes the list $[1, 2]$. Let $:_H$ be a function identical to $:$, but strict in the head field, and let H be the computable function on lists that replaces each $:$ by $:_H$. For example,

$$H\ (1 : 2 : \bot : 3 : []) \;=\; 1 :_H 2 :_H \bot :_H 3 :_H [] \;=\; 1 : 2 : \bot$$

Similarly, let $:_T$ be a function identical to $:$ but strict in the tail field, and let T be the computable function that replaces each $:$ by $:_T$.

We say a function f is *head strict* if it is safe to replace each $:$ by $:_H$ in the argument to f. In other words, f is head strict if $f = f \circ H$. Similarly, we say f is *tail strict* if $f = f \circ T$.

An example of a head strict function is *before*, which returns the segment of a list before the first zero (see Figure 1). Any cons cell examined by *before* will have its head examined to see if it is zero, hence *before* is head strict. If a bottom occurs before the first zero, we have, say,

$$\textit{before}\ (1 : \bot : 0 : []) \;=\; \textit{before}\ (1 : \bot) \;=\; 1 : \bot$$

$$
\begin{array}{lcl}
\mathit{before}\ xs & = & \textbf{case}\ \ xs\ \ \textbf{of} \\
 & & \quad [\,] \qquad \Rightarrow\ [\,] \\
 & & \quad y : ys\ \ \Rightarrow\ \textbf{if}\ \ y = 0 \\
 & & \qquad\qquad\qquad\ \ \textbf{then}\ \ [\,] \\
 & & \qquad\qquad\qquad\ \ \textbf{else}\ \ \ y : \mathit{before}\ ys
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{length}\ xs & = & \textbf{case}\ \ xs\ \ \textbf{of} \\
 & & \quad [\,] \qquad \Rightarrow\ 0 \\
 & & \quad y : ys\ \ \Rightarrow\ 1 + \mathit{length}\ ys
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{doubles}\ xs & = & \textbf{case}\ \ xs\ \ \textbf{of} \\
 & & \quad [\,] \qquad \Rightarrow\ [\,] \\
 & & \quad y : ys\ \ \Rightarrow\ (2 \times y) : \mathit{doubles}\ ys
\end{array}
$$

Figure 1: Example programs

which is as required, since $H\ (1 : \bot : 0 : [\,]) = 1 : \bot$. If a bottom occurs after the first zero, we have, say

$$\mathit{before}\,(1 : 2 : 0 : \bot : 3 : [\,]) \ = \ \mathit{before}\,(1 : 2 : 0 : \bot) \ = \ 1 : 2 : [\,]$$

which is again as required, since $H\ (1 : 2 : 0 : \bot : 3 : [\,]) = 1 : 2 : 0 : \bot$.

An example of a tail strict function is *length*, which finds the length of a list. Any cons cell examined by *length* will also have its tail examined, hence *length* is tail strict. If any tail field contains bottom, then *length* is undefined; for instance

$$\mathit{length}\ (1 : 2 : 3 : \bot) \ = \ \mathit{length}\ \bot \ = \ \bot$$

which is as required, since $T\ (1 : 2 : 3 : \bot) = \bot$. If no tail field contains bottom, the *length* is defined even if some head is bottom; for instance

$$\mathit{length}\ (1 : \bot : 3 : [\,]) \ = \ 3$$

which is again as required, since $T\ (1 : \bot : 3 : [\,]) = 1 : \bot : 3 : [\,]$.

This characterization of strictness is useful because it can enable important optimizations. Say we have a program containing a fragment of the form $f \circ g$, and we know that f is head strict. Then the fragment is equivalent to $f \circ H \circ g$. So we may replace this call of g by a call to a new version g_H of g, in which every $:$ operation that produces part of the result is replaced by $:_H$. The head arguments of these cons operations may be evaluated immediately, instead of constructing a graph to be evaluated later. This can lead to significant improvements in efficiency.

Thus, our goal is to label every sub-expression of the program with a context function like H or T that indicates what components of a structure can be evaluated immediately.

We will see later that contexts can also be used to indicate simple strictness information, such as what arguments in a function call need to be evaluated.

Contexts themselves provide useful information for propagating contexts further. Consider the function *doubles*, which doubles every element in a list. Clearly, *doubles* is not head strict. For instance

$$\mathit{doubles}\ (1 : \bot : 3 : [\,]) \ = \ 2 : \bot : 6 : [\,] \ \neq \ 2 : \bot \ = \ \mathit{doubles}\ (1 : \bot)$$

even though $H\ (1 : \bot : 3 : [\,]) = 1 : \bot$. So $\mathit{doubles} \neq \mathit{doubles} \circ H$. However, it is not hard to see that $H \circ \mathit{doubles} = H \circ \mathit{doubles} \circ H$, that is, *doubles* is head strict in a head strict context. Since *before* is head strict we have

$$\mathit{before} \circ \mathit{doubles} \ = \ \mathit{before} \circ H \circ \mathit{doubles} \ = \ \mathit{before} \circ H \circ \mathit{doubles} \circ H$$

showing that it is safe to replace $:$ by $:_H$ in the argument to *doubles* when the result is examined by a head strict function like *before*.

One reason that head strictness is of particular importance is that printing may induce a head strict context. Say that e is an expression that returns a list of characters to be printed on a terminal. Then e and $H\ e$ will both print exactly the same results, but the same cannot be said of e and $T\ e$. Thus, a character printer is head strict but not tail strict.

Traditionally, a function f is said to be *strict* if $f\ \bot = \bot$. Tail strictness can be characterized similarly: a strict function f is tail strict iff $f\ u = \bot$ whenever any tail of u is $\bot$. This holds because if any tail of u is $\bot$ then $T\ u = \bot$, so $f\ u = f\ (T\ u) = f\ \bot = \bot$. Similarly, a strict function f is both head strict and tail strict iff $f\ u = \bot$ whenever any head or tail of u is $\bot$. This "$f\ u = \bot$" approach was used in [Wad87].

However, it is *not* true that f is head strict only if $f\ u = \bot$ whenever some head of u is $\bot$. Two counter-examples appear above, where $\mathit{before}\ u \neq \bot$ although some head of u is $\bot$. A main advantage of the context approach is that it can describe head strictness, whereas the "$f\ u = \bot$" approach cannot.

2 Projections

A continuous function α is a *projection* [Sco81] if for every object u,

$$\begin{aligned} \alpha\ u &\sqsubseteq u \\ \alpha\,(\alpha\ u) &= \alpha\ u \end{aligned}$$

The first line says that projections only remove information from an object. The second line says that all the information is removed at once, so applying the projection a second time has no effect. These two properties can also be written

$$\begin{aligned} \alpha &\sqsubseteq \mathit{ID} \\ \alpha \circ \alpha &= \alpha \end{aligned}$$

where ID is the identity function, defined by $ID\ u = u$ for all u. In this paper, α, β, γ, etc., will always denote projections.

As we have seen, projections such as H and T characterize the context in which a value is needed, and so are useful for this style of strictness analysis. We will use the words "projection" and "context" interchangably.

Projections form a complete lattice under the $\sqsubseteq$ ordering, with ID at the top and BOT at the bottom, where BOT is the function defined by $BOT\ u = \perp$ for all u. For example, the projections we have seen so far form the following lattice,

$$\begin{array}{ccc} & ID \bullet & \\ H \bullet & & \bullet\ T \\ & H \sqcap T \bullet & \\ & BOT \bullet & \end{array}$$

The projection $H \sqcap T$ corresponds to being both head and tail strict.

We say that a function f is *β-strict in context α* if $\alpha \circ f = \alpha \circ f \circ \beta$, and write $f : \alpha \Rightarrow \beta$. For example, we have seen that $before : ID \Rightarrow H$ and $doubles : H \Rightarrow H$.

An alternate characterization of $f : \alpha \Rightarrow \beta$ is given by the following result.

Proposition: $f : \alpha \Rightarrow \beta$ iff $\alpha \circ f \sqsubseteq f \circ \beta$.

Proof: In the forward direction, since $\alpha \circ f = \alpha \circ f \circ \beta$ and $\alpha \sqsubseteq ID$, we may conclude $\alpha \circ f \sqsubseteq f \circ \beta$. In the reverse direction, composing α with each side gives $\alpha \circ \alpha \circ f \sqsubseteq \alpha \circ f \circ \beta$, and since $\alpha \circ \alpha = \alpha$, we have $\alpha \circ f \sqsubseteq \alpha \circ f \circ \beta$. Since $\beta \sqsubseteq ID$, we also have $\alpha \circ f \sqsupseteq \alpha \circ f \circ \beta$, which gives the desired equality. $\Box$

The strictness relation satisfies a useful composition result.

Proposition: If $f : \alpha \Rightarrow \beta$ and $g : \beta \Rightarrow \gamma$ then $f \circ g : \alpha \Rightarrow \gamma$.

Proof: Immediate, since $\alpha \circ f \circ g \sqsubseteq f \circ \beta \circ g \sqsubseteq f \circ g \circ \gamma$. $\Box$

For example, from $before : ID \Rightarrow H$ and $doubles : H \Rightarrow H$ we may conclude, as noted in the previous section, that $before \circ doubles : ID \Rightarrow H$.

A similar argument shows that if $\alpha \sqsubseteq \beta$ and $f : \beta \Rightarrow \gamma$ and $\gamma \sqsubseteq \delta$ then $f : \alpha \Rightarrow \delta$. In particular, $f : \alpha \Rightarrow ID$ for every f and α.

We now verify the assertion made above, that projections form a complete lattice. We first take care of the least upper bound.

Proposition: If A is a set of projections, then $\bigsqcup A$ exists and is a projection.

Proof: For every $\alpha \in A$ we have $\alpha \sqsubseteq ID$, so clearly $\bigsqcup A$ exists and $\bigsqcup A \sqsubseteq ID$. Further, from this it follows that $\bigsqcup A \circ \bigsqcup A \sqsubseteq \bigsqcup A$, so it remains to show that $\bigsqcup A \circ \bigsqcup A \sqsupseteq \bigsqcup A$.

Then for every object u we have

$$
\begin{array}{ll}
\multicolumn{2}{l}{\bigsqcup A\ (\bigsqcup A\ u)} \\
= & \bigsqcup A\ (\bigsqcup\{\alpha\ u \mid \alpha \in A\}) \\
\sqsupseteq & \bigsqcup\{\bigsqcup A\ (\alpha\ u) \mid \alpha \in A\} \\
\sqsupseteq & \bigsqcup\{\alpha\ (\alpha\ u) \mid \alpha \in A\} \\
= & \bigsqcup\{\alpha\ u \mid \alpha \in A\} \\
= & \bigsqcup A\ u
\end{array}
$$

as required. $\square$

Now, a difficulty arises. If we let $\alpha \sqcap \beta$ denote the largest continuous function f such that $f \sqsubseteq \alpha$ and $f \sqsubseteq \beta$, then f may not be a projection. (Counter-example: Consider the domain $\{a, b, c\}$ with $a \sqsubset b \sqsubset c$. Let $\alpha = \{a \mapsto a, b \mapsto a, c \mapsto c\}$ and $\beta = \{a \mapsto a, b \mapsto b, c \mapsto b\}$. Then $f = \{a \mapsto a, b \mapsto a, c \mapsto b\}$, which is not a projection since $f\ c = b$ but $f\ (f\ c) = a$.)

Therefore, we adopt the convention that $\alpha \sqcap \beta$ denotes the largest *projection* γ such that $\gamma \sqsubseteq \alpha$ and $\gamma \sqsubseteq \beta$. (In the counter-example above, we would have $\gamma = \{a \mapsto a, b \mapsto a, c \mapsto a\}$.) With this convention, the greatest lower bound of a set of projections is given by

$$\sqcap A \ = \ \bigsqcup\{\beta \mid \text{for all } \ \alpha \in A, \beta \sqsubseteq \alpha\}$$

where β ranges over projections. It follows from the above proposition that $\sqcap A$ exists and is a projection.

3 Strictness and absence

We have used projections to characterize such exotic concepts as head strictness and tail strictness, but we have not yet tackled ordinary strictness, defined by $f\ \bot\ =\ \bot$. Using projections to characterize strictness is possible, but requires some extensions to the framework described so far.

Roughly speaking, the problem is that projections, as described so far, let us specify what information is *sufficient* but not what information is *necessary*. For example, say that f is head strict, that is, $f\ = f \circ H$. Then we know that if the argument of f is, say, $1 : \bot : []$ then it is *sufficient* to use the value $H\ (1 : \bot : []) = 1 : \bot$ instead. But to characterize strictness, we must say something about what information is *necessary*. In particular, we must use projections to say that it is necessary that a value be more defined than $\bot$.

In order to specify information about necessity with projections, we extend our domains with a new element $\lightning$, pronounced "abort" (the symbol is intended to resemble a lightning bolt). The interpretation of $\alpha\ u = \lightning$ will be that α requires a value more defined than u. In order for this interpretation to work, we require that all functions be strict in $\lightning$, that is, $f\ \lightning = \lightning$ for all functions f. Intuitively, if a value is not acceptable it is mapped into $\lightning$, which causes all computation to abort immediately. To define strictness, we will use a projection STR that does not accept $\bot$, so we must have $STR\ \bot = \lightning$. Since any projection must satisfy $\alpha\ u \sqsubseteq u$, we must have $\lightning$ beneath $\bot$ in the domain ordering.

Therefore we extend each domain D to a new domain $D_{\lightning}$, derived by lifting D and adding a new bottom element, $\lightning$, beneath the existing bottom, $\bot$. So $\lightning \sqsubset \bot \sqsubseteq u$, for every $u \in D$. Every function $f : D_1 \to D_2$ is extended to a function $f : D_{1\lightning} \to D_{2\lightning}$ by making f strict in $\lightning$. All functions are strict in $\lightning$, but may or may not be strict in $\bot$. In particular, cons is strict in $\lightning$, so $(u : \lightning) = \lightning = (\lightning : v)$ for all u and v.

(A technical point: since *everything*, even the conditional, is strict in $\lightning$, the least fixpoint of any recursive function definition is the constant $\lightning$ function. This is not what we want. For recursive functions definitions in our language, we take the least fixpoint above the function BOT defined by $BOT\ u = \bot$ if $\lightning \sqsubset u$, and $BOT\ \lightning = \lightning$.)

The extended domains allow contexts to specify information about necessity. A value u is *unacceptable* to a context α if $\alpha\ u = \lightning$.

Proposition: If $f : \alpha \Rightarrow \beta$ and u is unacceptable to β, then $f\ u$ is unacceptable to α.

Proof: Assuming $\alpha \circ f = \alpha \circ f \circ \beta$ and $\beta\ u = \lightning$ gives

$$\alpha\,(f\ u)\ =\ \alpha\,(f\ (\beta\ u))\ =\ \alpha\,(f\ \lightning)\ =\ \lightning$$

as required. $\square$

The projection STR is defined by setting

$$\begin{array}{lcll} STR\ \lightning & = & \lightning & \\ STR\ \bot & = & \lightning & \\ STR\ u & = & u & \text{if } \bot \sqsubset u \end{array}$$

We can now capture the notion of strictness precisely.

Proposition: $f : STR \Rightarrow STR$ iff f is strict.

Proof: In the forward direction, the only value unacceptable to STR is $\bot$, so it follows from the preceding result that $u = \bot$ implies $f\ u = \bot$, so f is strict. In the backward direction, we must show that if f is strict then $STR\ (f\ u) \sqsubseteq f\ (STR\ u)$ for all u. If $u \neq \bot$ this follows since $STR\ u = u$. If $u = \bot$ this follows since both sides of the inequality reduce to $\lightning$: on the left $STR\ (f\ \bot) = STR\ \bot = \lightning$ and on the right $f\ (STR\ \bot) = f\ \lightning = \lightning$. $\square$

Although ID is still the top element of the domain of projections, BOT is no longer the bottom. The new bottom element is the projection $FAIL$, defined by $FAIL\ u = \lightning$ for all u. Say that a function $f : D_1 \to D_2$ is *divergent* if $f\ u = \bot$ for every $u \in D_1$. The following two results are of interest.

Proposition: $f : STR \Rightarrow FAIL$ iff f is divergent.

Proposition: $g : FAIL \Rightarrow FAIL$ for every function g.

Both proofs are simple exercises. As a corollary, the composition rule implies that if f is divergent, then so is $f \circ g$ for any g.

The old projection BOT is rechristened ABS, for "absent", and defined by

$$\begin{array}{lcll} ABS\ \lightning & = & \lightning & \\ ABS\ \perp & = & \perp & \\ ABS\ u & = & \perp & \text{if } \perp \sqsubset u \end{array}$$

Say that a function $f : D_1 \rightarrow D_2$ *ignores its argument* if $f\ u = f\ \perp$ for every $u \in D_1$. Again, there are two results of interest.

Proposition: $f : STR \Rightarrow ABS$ iff f ignores its argument.

Proposition: $g : ABS \Rightarrow ABS$ for every function g.

And again, both proofs are simple exercises. As a corollary, the composition rule implies that if f ignores its argument, then so does $f \circ g$ for any g.

The four projections we have been discussing have the following domain ordering:

$$\begin{array}{ccc} & ID\ \bullet & \\ ABS\ \bullet & & \bullet\ STR \\ & FAIL\ \bullet & \end{array}$$

This is just a *subdomain* of the domain of projections over D. This is discussed more fully in the next section.

Context analysis can yield useful information for a compiler. If analysis succeeds in labelling a sub-expression e with one of these contexts, then the following interpretations apply:

- *FAIL*. No value that could be returned by e is acceptable. The compiler may safely implement e by code that aborts the program immediately.

- *ABS*. The value of e is ignored. The compiler may safely implement e by code that returns a dummy value.

- *STR*. The value of e is required. The compiler may safely implement e by code that evaluates e immediately; no graph for e need be constructed.

- *ID*. The value of e may be required or may be ignored. The compiler can safely implement e only by constructing a graph.

Strictness annotations having exactly these four meanings are used in Wray's strictness analyzer [Wra86, FW86]. Most strictness analysers only distinguish between strict and non-strict arguments, corresponding to the distinction between STR and ID above. The compilation possibilities afforded by $FAIL$ and ABS are extra optimisations, not available to most compilers.

We will call $\alpha : D_{\lightning} \to D_{\lightning}$ a projection *over* D. Strictly speaking, we should write ID_D, STR_D, ABS_D, and $FAIL_D$ to indicate the domain D that these projections are over. Usually we will omit the domain subscript since it can be derived, as the phrase goes, from context.

The *strict part* of a projection α is $\alpha' = \alpha \sqcap STR$. For example, $ID' = STR$ and $ABS' = FAIL$. A context α is *strict* if it is equal to its strict part, or equivalently, if $\alpha \sqsubseteq STR$. Ironically, this implies that α is called strict iff $\alpha \perp = \lightning$, and non-strict iff $\alpha \perp = \perp$.

The problem of analysing strictness in context α can be reduced to analysing strictness in context α':

Proposition: If α is non-strict and $f : \alpha' \Rightarrow \beta$ then $f : \alpha \Rightarrow ABS \sqcup \beta$.

Proof: Since α is non-strict, $\alpha = ABS \sqcup \alpha'$ and we have

$$(ABS \sqcup \alpha') \circ f \;=\; (ABS \circ f) \sqcup (\alpha' \circ f) \;\sqsubseteq\; (f \circ ABS) \sqcup (f \circ \beta) \;\sqsubseteq\; f \circ (ABS \sqcup \beta)$$

as required. $\square$

For instance, say f is a strict function, so $f : STR \Rightarrow STR$. That is, if the result of f is needed, then the argument of f will be needed. Then since $ID' = STR$ and $ABS \sqcup STR = ID$, from the above we have $f : ID \Rightarrow ID$. That is, if the result of f may or may not be needed, then the argument of f may or may not be needed.

4 Finite domains

Let INT be the flat domain of integers. Clearly, there are an infinite number of projections α over INT. Using these we can specify quite precise information about functions. For example, for any integer m let $EQUAL_m$ be the projection defined by

$$\begin{array}{lcll} EQUAL_m\ m & = & m & \\ EQUAL_m\ u & = & \lightning & \text{if } u \neq m \end{array}$$

Then if $f : EQUAL_m \Rightarrow EQUAL_n$ it follows that $f\ n = m$.

For some applications this expressiveness may be useful, but it is more precise than required for strictness analysis in a compiler. Fortunately, we need not maintain such precise information. The purpose of context analysis is to label each sub-expression e in a program with a context α such that $\alpha\ e$ and e return the same result (within the larger context in which the sub-expression appears). Clearly, if it is safe to label e with α then it is also safe to label e with any β such that $\alpha \sqsubseteq \beta$. For example, we can safely approximate $EQUAL_m$ by STR, for any m.

This gives a notion of "approximation" inverted from the usual one. The inequality $\alpha \sqsubseteq \beta$ is traditionally read "α approximates β", meaning $\alpha\ u$ is less defined than $\beta\ u$ for every u. However, we may also read it as "α is approximated by β", meaning α conveys more precise information than β about the values acceptable in some context.

Thus for purposes of strictness analysis in a compiler, we may choose to use any subset of projections, so long as it is closed under the operations of interest (e.g., $\sqcup$ and $\sqcap$) and so long as it contains the largest projection, ID. For analyzing flat domains, such as the integers, a good choice is the four point subdomain of projections, $\{ID, STR, ABS, FAIL\}$, discussed in the preceding section.

The subset of projections that we choose need not even include $FAIL$. For example, the two point domain $\{ID, STR\}$ can provide quite useful analyses. When using this two point domain, ABS is approximated by ID, and $FAIL$ is approximated by STR. This is indeed safe: ID safely approximates everything, since it is always safe to construct a graph; and STR safely approximates $FAIL$, since if the function is going to diverge anyway (as for $FAIL$) it is safe to evaluate the argument in advance (as for STR).

5 Finite domains for lists

Let $LIST\ D$ be the non-flat domain of lists whose elements are in domain D. For example, $LIST\ INT$ is the domain of lists of integers, and $LIST\ (LIST\ INT)$ is the domain of lists of lists of integers. We have already discussed two projections over $LIST\ D$, namely H and T. This section presents finite domains of projections over $LIST\ D$, analogous to the finite domains of projections over INT presented in the previous section.

It is convenient to define the projection NIL and the projection generator $CONS$. The projection NIL over $LIST\ D$ is defined by

$$\begin{array}{lcl} NIL\ \lightning & = & \lightning \\ NIL\ \bot & = & \lightning \\ NIL\ [\,] & = & [\,] \\ NIL\ (u : v) & = & \lightning \end{array}$$

If α is a projection over D, and β is a projection over $LIST\ D$, the projection $CONS\ \alpha\ \beta$ on $LIST\ D$ is defined by

$$\begin{array}{lcl} CONS\ \alpha\ \beta\ \lightning & = & \lightning \\ CONS\ \alpha\ \beta\ \bot & = & \lightning \\ CONS\ \alpha\ \beta\ [\,] & = & \lightning \\ CONS\ \alpha\ \beta\ (u : v) & = & \alpha\ u : \beta\ v \end{array}$$

These projections can be used to describe lists precisely. For example,

$$CONS\ ID\ (CONS\ EQUAL_0\ NIL)$$

specifies the context that only accepts lists of length two whose second element is zero.

Of special interest are projections which treat all elements of a list in the same way. If α is a projection over D, then the projections $FIN\ \alpha$ and $INF\ \alpha$ over $LIST\ D$ are defined by

$$FIN\ \alpha = NIL \sqcup CONS\ \alpha\ (FIN\ \alpha)$$
$$INF\ \alpha = NIL \sqcup CONS\ \alpha\ (ABS \sqcup INF\ \alpha)$$

Roughly speaking, $FIN\ \alpha$ accepts only finite lists, each element of which is accepted by α, and $INF\ \alpha$ accepts finite or infinite lists, each element of which is accepted by α. Neither accepts $\perp$, so $FIN\ \alpha$ and $INF\ \alpha$ are strict for every α, even if α is non-strict.

These projections are related to the ones discussed previously by the following equations:

$$\begin{array}{rcl@{\qquad}rcl}
STR & = & INF\ ID & ID & = & ABS \sqcup INF\ ID \\
H' & = & INF\ STR & H & = & ABS \sqcup INF\ STR \\
T' & = & FIN\ ID & T & = & ABS \sqcup FIN\ ID \\
H' \sqcap T' & = & FIN\ STR & H \sqcap T & = & ABS \sqcup FIN\ STR
\end{array} \qquad (*)$$

The projections H and T were defined before $\lightning$ was introduced, so $H\ \perp = \perp$ and $T\ \perp = \perp$, and therefore H and T are non-strict. The corresponding strict versions are, of course, H' and T'.

As another example, we have $length : STR \Rightarrow FIN\ ABS$. That is, *length* is defined only for finite lists, but the elements of the list are ignored.

Let D_C be a finite domain of projections over D. For example, if D is INT then D_C might be $\{STR, ID\}$. The finite domain $LIST_C\ D_C$ of projections over $LIST\ D$ consists of the projections

$$\begin{array}{l@{\qquad}l}
INF\ \alpha & ABS \sqcup INF\ \alpha \\
FIN\ \alpha & ABS \sqcup FIN\ \alpha
\end{array}$$

for each $\alpha \in D_C$, plus $FAIL$ and ABS. Note that STR and ID are implicitly included in $LIST_C\ D_C$, since from above we have $STR = INF\ ID$ and $ID = ABS \sqcup INF\ ID$, and ID must be in D_C.

For example, if D_C is the two point domain $\{ID, STR\}$, then $LIST_C\ D_C$ is the ten point domain consisting of the eight projections in $(*)$ plus ABS and $FAIL$. A diagram of this domain appears in Figure 2.

As a second example, if D_C is the four point domain $\{ID, STR, ABS, FAIL\}$, then $LIST_C\ D_C$ is a domain with sixteen points. One would expect eighteen points $(4 \times 4 + 2)$, but

$$\begin{array}{rcl}
FIN\ FAIL & = & INF\ FAIL \\
ABS \sqcup FIN\ FAIL & = & ABS \sqcup INF\ FAIL
\end{array}$$

and so four of the points collapse to two. The identifications arise because $CONS\ FAIL\ \alpha = FAIL$ for any projection α, and so $FIN\ FAIL = INF\ FAIL = NIL$.

6 Context analysis

The problem of context analysis is this: given a program defining f and a projection α, we wish to find a projection β such that $f : \alpha \Rightarrow \beta$. Of course, we could just always take β to be ID, but if possible we would like to find a smaller projection.

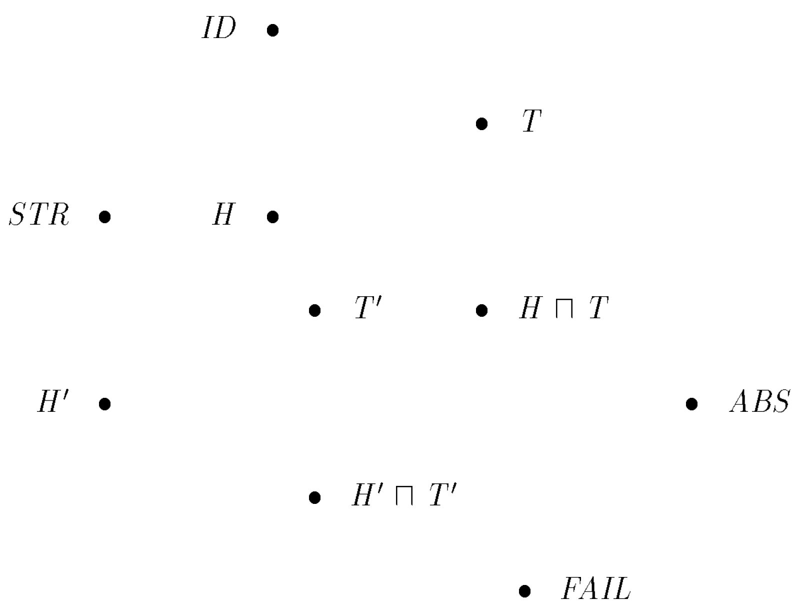

Figure 2: A finite domain of projections for lists

Ideally, given f and α we would like to find the *smallest* β such that $f : \alpha \Rightarrow \beta$. There are two difficulties with this. First, it is not clear that a smallest β always exists. Second, even if it did exist, it would not be computable. As we have already observed, $f : STR \Rightarrow FAIL$ holds iff f diverges for every argument; so if we could always find the smallest β then we could solve the halting problem. Therefore we will have to settle for finding *some* β, not necessarily the smallest one.

6.1 Language

To analyse a function f we will need to examine the program that defines it. We will use a small first-order language, with the following grammar:

$$
\begin{array}{lcll}
e & ::= & x & \text{variables} \\
 & \mid & k & \text{constants} \\
 & \mid & f\ e_1\ \ldots\ e_n & \text{function applications} \\
 & \mid & \textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & \text{conditionals} \\
 & \mid & \textbf{case}\ e_0\ \textbf{of}\ [\,] \Rightarrow e_1 \mid y : ys \Rightarrow e_2 & \text{case expressions}
\end{array}
$$

Each function f has a fixed arity n. Function definitions have the form

$$f\ x_1\ \ldots\ x_n \quad = \quad e$$

Infixes are allowed as usual; $e_1 + e_2$ is equivalent to $(+)\ e_1\ e_2$, where $(+)$ is a function name. Some programs in this language are shown in Figure 1 of Section 1 and Figure 5 of Section 7.

6.2 Projection transformers

For each function f of n arguments, and each i from 1 to n, we will define f^i to be a transformer that takes a projection applied to the result of f into a projection that may safely be applied to the i'th argument. That is, f^i must satisfy the following *safety requirement*: if $\beta_i = f^i\ \alpha$ then

$$\alpha\ (f\ u_1 \ldots u_i \ldots u_n)\ \sqsubseteq\ f\ u_1 \ldots (\beta_i\ u_i) \ldots u_n$$

for all $u_1, \ldots, u_n$. In particular, if f is a function of one argument, and $\beta = f^1\ \alpha$, then the safety requirement ensures that $f : \alpha \Rightarrow \beta$.

It is easy to show that the safety requirements for $f^1, \ldots, f^n$ are satisfied iff

$$\alpha\ (f\ u_1 \ldots u_n)\ \sqsubseteq\ f\ (\beta_1\ u_1) \ldots (\beta_n\ u_n)$$

for all $u_1, \ldots, u_n$, where $\beta_i = f^i\ \alpha$ for each i from 1 to n.

Similar to f^i, for each expression e and each variable x, we will define e^x to be a transformer that takes a projection applied to e into a projection that may safely be applied to each instance of x in e. That is, e^x must satisfy the safety requirement: if $\beta = e^x\ \alpha$ then

$$\alpha\ e\ \sqsubseteq\ e[(\beta\ x)/x]$$

for all values of the variables in e (including x). Here, as usual, $e_0[e_1/x]$ denotes the result of substituting e_1 for each instance of x in e_0.

(To make this definition more formal, we should give a semantics of the language, defining $E[\![e]\!]\rho$ for each expression e and environment ρ. The safety requirement becomes that if $\beta = e^x\ \alpha$ then

$$\alpha\ (E[\![e]\!]\rho)\ \sqsubseteq\ E[\![e]\!](\rho[(\beta\ (\rho[\![x]\!]))/[\![x]\!]])$$

for each environment ρ. To be more formal still, for $e^x\ \alpha$ we should write something like $M[\![e]\!][\![x]\!]\alpha$.)

Having specified the safety conditions, we must now give definitions of f^i and e^x satisfying these conditions. Definitions of f^i for primitive f appear in Section 6.7. Otherwise, if the program defining f is

$$f\ x_1 \ldots x_n\ =\ e$$

then f^i is defined by

$$f^i\ \alpha\ =\ e^{x_i}\ \alpha$$

for each i from 1 to n. The definition of e^x may in turn refer to the f^i, so the definitions are mutually recursive. The full definition of e^x is given in Figure 3. If it looks forbidding, don't worry: all will be explained as we go along.

It is clear that the rule defining f^i is safe (that is, satisfies the safety condition) if the rules defining e^x are safe. In what follows, we will show that the rules defining e^x are safe if the f^i are safe; and that the f^i are safe for primitive f. It follows by recursion induction that the definitions of f^i and e^x are indeed safe.

$$
\begin{array}{lll}
e^x\ \alpha & = & \alpha \rhd e^x\ \alpha'
\end{array}
$$

If α is strict and $\alpha \neq \mathit{FAIL}$ then:

$$
\begin{array}{llll}
x^x\ \alpha & = & \alpha & \\
y^x\ \alpha & = & \mathit{ABS} & \text{if } x \neq y \\
k^x\ \alpha & = & \mathit{ABS} & \text{if } k \text{ is a constant} \\
(f\ e_1 \ldots e_n)^x\ \alpha & & & \\
& = & e_1^x\ (f^1\ \alpha)\ \&\ \cdots\ \&\ e_n^x\ (f^n\ \alpha) & \\
(\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2)^x\ \alpha & & & \\
& = & e_0^x\ \mathit{STR}\ \&\ (e_1^x\ \alpha \sqcup e_2^x\ \alpha) & \\
(\textbf{case}\ e_0\ \textbf{of}\ [\,] \Rightarrow e_1 \mid y : ys \Rightarrow e_2)^x\ \alpha & & & \\
& = & (e_0^x\ \mathit{NIL}\ \&\ e_1^x\ \alpha) \sqcup (e_0^x\ (\mathit{CONS}\ (e_2^y\ \alpha)\ (e_2^{ys}\ \alpha))\ \&\ e_2^x\ \alpha) &
\end{array}
$$

Figure 3: Definition of $e^x\ \alpha$

Three rules in the definition of e^x are obvious:

$$
\begin{array}{llll}
x^x\ \alpha & = & \alpha & \\
y^x\ \alpha & = & \mathit{ABS} & \text{if } x \neq y \\
k^x\ \alpha & = & \mathit{ABS} & \text{if } k \text{ is a constant}
\end{array}
$$

More generally, it is safe to set $e^x\ \alpha = \mathit{ABS}$ whenever x does not appear in e.

As an example, say that the constant function K is defined by

$$K\ x\ y \quad = \quad x$$

Then we have $K^1\ \alpha = x^x\ \alpha = \alpha$ and $K^2\ \alpha = x^y\ \alpha = \mathit{ABS}$. In other words, evaluating K in context α causes its first argument to be evaluated in context α and its second argument to be ignored.

6.3 The $\rhd$ operation

Results from Section 3 guarantee that it is safe to set $e^x\ \mathit{FAIL} = \mathit{FAIL}$ and $e^x\ \mathit{ABS} = \mathit{ABS}$, and that if α is non-strict we may set $e^x\ \alpha = \mathit{ABS} \sqcup e^x\ \alpha'$. Therefore, in defining $e^x\ \alpha$, we can restrict our attention to the case that α is strict and not FAIL.

To aid in doing so, we introduce the $\rhd$ operator, pronounced "guard", and defined by

$$
\begin{array}{llll}
\mathit{FAIL} \rhd \beta & = & \mathit{FAIL} & \\
\mathit{ABS} \rhd \beta & = & \mathit{ABS} & \\
\alpha \rhd \beta & = & \beta & \text{if } \alpha \text{ is strict and } \alpha \neq \mathit{FAIL} \\
(\mathit{ABS} \sqcup \alpha) \rhd \beta & = & \mathit{ABS} \sqcup \beta & \text{if } \alpha \text{ is strict and } \alpha \neq \mathit{FAIL}
\end{array}
$$

It follows from the above that we may safely set

$$e^x\ \alpha \quad = \quad \alpha \rhd e^x\ \alpha'$$

This rule holds for all α, but has no effect unless α is non-strict or FAIL. We assume in all other rules that α is strict and not FAIL.

6.4 Application and the & operation

Say we wish to determine $(f\ e_1\ e_2)^x\ \alpha$. Then we need to find a δ such that

$$\alpha\ (f\ e_1\ e_2)\ \sqsubseteq\ (f\ e_1\ e_2)[\delta\ x/x]$$

We go about doing so in three stages.

First, we know from the definition of f^i that

$$\alpha\ (f\ e_1\ e_2)\ \sqsubseteq\ f\ (\beta_1\ e_1)\ (\beta_2\ e_2)$$

where $\beta_i = f^i\ \alpha$, for $i = 1, 2$. Second, we know from the definition of e_i^x that

$$f\ (\beta_1\ e_1)\ (\beta_2\ e_2)\ \sqsubseteq\ f\ (e_1[\gamma_1\ x/x])\ (e_2[\gamma_2\ x/x])$$

where $\gamma_i = e_i^x\ \beta_i$, for $i = 1, 2$. Third, we need to find a δ such that

$$f\ (e_1[\gamma_1\ x/x])\ (e_2[\gamma_2\ x/x])\ \sqsubseteq\ (f\ e_1\ e_2)[\delta\ x/x]$$

Clearly, we could take $\delta = \gamma_1 \sqcup \gamma_2$. But we can do a little better than this. Since all functions are strict in $\lightning$, assuming x appears in both e_1 and e_2 then if either $\gamma_1\ x = \lightning$ or $\gamma_2\ x = \lightning$ the left hand side evaluates to $\lightning$, so we may safely set $\delta\ x = \lightning$. Therefore we define

$$(\gamma_1 \mathbin{\&} \gamma_2)\ u\ =\ \begin{cases} \lightning & \text{if}\ \ \gamma_1\ u = \lightning\ \text{or}\ \ \gamma_2\ u = \lightning \\ \gamma_1\ u \sqcup \gamma_2\ u & \text{otherwise} \end{cases}$$

Taking $\delta = \gamma_1 \mathbin{\&} \gamma_2$, we have shown that the rule

$$(f\ e_1\ e_2)^x\ \alpha\ =\ e_1^x\ (f^1\ \alpha) \mathbin{\&} e_2^x\ (f^2\ \alpha)$$

is safe. For a function of n arguments, we get the rule show in Figure 3.

What if x does not appear in e_1 or e_2? No problem. If x does not appear in e_i, then $\gamma_i = e_i^x\ \beta_i = ABS$, so we won't have $\gamma_i\ x = \lightning$ anyhow (unless x is $\lightning$, of course).

As an example, we have

$$\begin{aligned}(K\ x\ y)^y\ STR\ &=\ x^y\ (K^1\ STR) \mathbin{\&} y^y\ (K^2\ STR) \\ &=\ x^y\ STR \mathbin{\&} y^y\ ABS\ =\ ABS \mathbin{\&} ABS\ =\ ABS\end{aligned}$$

so the expression $K\ x\ y$ ignores the value of y, as we would expect.

The & operation satisfies many laws, some of which are shown in Figure 4: it is commutative, associative, idempotent, has ABS as a unit, $FAIL$ as a zero, distributes over $\sqcup$, and satisfies various properties with NIL and $CONS$.

Many people at first expect the role of & to be played by $\sqcap$. It is worth noting, therefore, that & is certainly different from $\sqcap$; for instance, $ABS \mathbin{\&} STR = STR$ whereas $ABS \sqcap STR = FAIL$. As we shall see in Section 8, Dybjer's inverse image analysis [Dyb87] might be considered a restriction to the class of projections for which & and $\sqcap$ are identical.

$$
\begin{array}{rcl}
\multicolumn{3}{c}{\&\text{ is commutative and associative}} \\
\alpha \mathbin{\&} \alpha & = & \alpha \\
\mathit{ABS} \mathbin{\&} \alpha & = & \alpha \\
\mathit{FAIL} \mathbin{\&} \alpha & = & \mathit{FAIL} \\[1ex]
\alpha \mathbin{\&} (\beta \sqcup \gamma) & = & (\alpha \mathbin{\&} \beta) \sqcup (\alpha \mathbin{\&} \gamma) \\[1ex]
\mathit{CONS}\ \alpha\ \beta \mathbin{\&} \mathit{NIL} & = & \mathit{FAIL} \\
\mathit{CONS}\ \alpha\ \beta \mathbin{\&} \mathit{CONS}\ \gamma\ \delta & = & \mathit{CONS}\ (\alpha \mathbin{\&} \gamma)\ (\beta \mathbin{\&} \delta)
\end{array}
$$

Figure 4: Laws of &

6.5 Conditional expressions

To derive the rule for conditional expressions, we need to find a δ such that

$$\alpha\,(\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2) \sqsubseteq (\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2)[\delta\ x/x]$$

Again, the derivation proceeds by a sequence of steps.

We begin by pointing out an *incorrect* derivation. An obvious first step would be

$$\alpha\,(\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2) = (\textbf{if}\ \mathit{STR}\ e_0\ \textbf{then}\ \alpha\ e_1\ \textbf{else}\ \alpha\ e_2)$$

where α is strict and STR is over booleans. But in a domain containing $\lightning$ this law is *invalid*. For example, if α is STR over integers, e_0 is true, e_1 is 1, and e_2 is $\perp$, then the left-hand side of the above equation yields 1, while the right-hand side yields $\lightning$ (because **if** true **then** 1 **else** $\lightning$ evaluates to $\lightning$ and not 1).

Instead, as a first step we use the rule

$$
\begin{array}{l}
\alpha\,(\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2) \\
\quad = \ (\textbf{if}\ \mathit{STR}\ e_0\ \textbf{then}\ \alpha\ e_1\ \textbf{else}\ \perp) \sqcup (\textbf{if}\ \mathit{STR}\ e_0\ \textbf{then}\ \perp\ \textbf{else}\ \alpha\ e_2)
\end{array}
$$

where, again, α is strict and STR is over booleans. It is easy to verify that this rule is valid by considering the four possibilities $\lightning$, $\perp$, true, and false for the value of e_0.

We then have

$$
\begin{array}{cl}
\multicolumn{2}{l}{\alpha\,(\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2)} \\
= & (\textbf{if}\ \mathit{STR}\ e_0\ \textbf{then}\ \alpha\ e_1\ \textbf{else}\ \perp) \sqcup \\
 & (\textbf{if}\ \mathit{STR}\ e_0\ \textbf{then}\ \perp\ \textbf{else}\ \alpha\ e_2) \\
\sqsubseteq & (\textbf{if}\ e_0[\beta_0\ x/x]\ \textbf{then}\ e_1[\beta_1\ x/x]\ \textbf{else}\ \perp) \sqcup \\
 & (\textbf{if}\ e_0[\beta_0\ x/x]\ \textbf{then}\ \perp\ \textbf{else}\ e_2[\beta_2\ x/x]) \\
\sqsubseteq & (\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ \perp)[((\beta_0 \mathbin{\&} \beta_1)\ x)/x] \sqcup \\
 & (\textbf{if}\ e_0\ \textbf{then}\ \perp\ \textbf{else}\ e_2)[((\beta_0 \mathbin{\&} \beta_2)\ x)/x] \\
\sqsubseteq & ((\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ \perp) \sqcup (\textbf{if}\ e_0\ \textbf{then}\ \perp\ \textbf{else}\ e_2)) \\
 & [(((\beta_0 \mathbin{\&} \beta_1) \sqcup (\beta_0 \mathbin{\&} \beta_2))\ x)/x] \\
= & (\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2)\ [(((\beta_0 \mathbin{\&} \beta_1) \sqcup (\beta_0 \mathbin{\&} \beta_2))\ x)/x] \\
= & (\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2)\ [((\beta_0 \mathbin{\&} (\beta_1 \sqcup \beta_2))\ x)/x]
\end{array}
$$

where $\beta_0 = e_0^x\ STR$, $\beta_1 = e_1^x\ \alpha$, and $\beta_2 = e_2^x\ \alpha$. The last step uses the distributive law from Figure 4.

This establishes the rule

$$(\textbf{if}\ \ e_0\ \ \textbf{then}\ \ e_1\ \ \textbf{else}\ \ e_2)^x\ \alpha \quad = \quad e_0^x\ STR \mathbin{\&} (e_1^x\ \alpha \sqcup e_2^x\ \alpha)$$

when α is strict. An intuitive reading of this rule is as follows. If the conditional expression is evaluated under strict α then x will be evaluated in e_0 under STR and either x will be evaluated in e_1 under α or x will be evaluated in e_2 under α.

Readers familiar with strictness analysis by abstract interpretation will note a resemblance between the rule given above and the rule

$$(\textbf{if}\ \ e_0\ \ \textbf{then}\ \ e_1\ \ \textbf{else}\ \ e_2)^\# \quad = \quad e_0^\# \sqcap (e_1^\# \sqcup e_2^\#)$$

used in abstract interpretation.

6.6 Case expressions

The rule for case expressions is

$$\begin{array}{ll} (\textbf{case}\ \ e_0\ \ \textbf{of}\ \ [\,] \Rightarrow e_1 \mid y : ys \Rightarrow e_2)^x\ \alpha & \\ \quad = \ (e_0^x\ NIL \mathbin{\&} e_1^x\ \alpha) \sqcup (e_0^x\ (CONS\ (e_2^y\ \alpha)\ (e_2^{ys}\ \alpha)) \mathbin{\&} e_2^x\ \alpha) & \end{array}$$

when α is strict. An intuitive reading of this rule is as follows. If the case expression is evaluated under strict α, then e_0 must evaluate to nil or to a cons cell. If e_0 evaluates to nil, then x will be evaluated in e_0 under NIL, and in e_1 under α. If e_0 evaluates to a cons cell, then the head of this cons cell will be evaluated as much as y is evaluated in e_2 under α, that is, the head will be evaluated under $e_2^y\ \alpha$. Similarly, the tail will be evaluated under $e_2^{ys}\ \alpha$. So x will be evaluated in e_0 under $(CONS\ (e_2^y\ \alpha)\ (e_2^{ys}\ \alpha))$, and in e_2 under α.

Using the concepts developed in the preceding sections, the proof of safety of this rule is straightforward, although lengthy. We outline only some key points here. To start, translate the case expression

$$\textbf{case}\ \ e_0\ \ \textbf{of}\ \ [\,] \Rightarrow e_1 \mid y : ys \Rightarrow e_2$$

to the equivalent form

$$\textbf{if}\ \ null\ e_0\ \ \textbf{then}\ \ e_1\ \ \textbf{else}\ \ e_2[head\ zs/y, tail\ zs/ys][e_0/zs]$$

where zs is a new variable and $null$, $head$, and $tail$ are defined in the usual way. The proof uses the facts

$$\begin{array}{lcl} \alpha\ (head\ e) & = & head\ ((CONS\ \alpha\ ABS)\ e) \\ \alpha\ (tail\ e) & = & tail\ ((CONS\ ABS\ \alpha)\ e) \end{array}$$

and the last two laws in Figure 4.

As pointed out in [Wad87], including case expressions in the language is essential when the analysis uses finite domains for lists. If the case expressions were rewritten in terms of *head* and *tail*, as above, then context analysis would yield less precise results. Case expressions are essential because they gather in one place information about how both the head and tail of the list are evaluated.

6.7 Primitive functions

Finally, we need to define f^i where f is a primitive function.

If f is strict in all its arguments, then we may set

$$f^i\ \alpha \quad = \quad \alpha \rhd STR$$

This definition is suitable for all primitives functions on flat domains, such as $(+)$ and $(=)$ over integers.

Most functions on non-flat domains, such as $(=)$ over lists, need not be given as primitives, since they can be defined in the language. The exception is the constructor function, $(:)$. Writing $HEAD$ for $(:)^1$ and $TAIL$ for $(:)^2$ (do not confuse these with H and T), we must have

$$\alpha\ (u : v) \ \sqsubseteq\ ((HEAD\ \alpha)\ u) : ((TAIL\ \alpha)\ v)$$

for all u, v, and α. It is not hard to verify that the following definitions fit the bill:

$$\begin{array}{lcl} (HEAD\ \alpha)\ u & = & \bigsqcup_{v \in LIST\ D}\ head\ (\alpha\ (u : v)) \\ (TAIL\ \alpha)\ v & = & \bigsqcup_{u \in D}\ tail\ (\alpha\ (u : v)) \end{array}$$

where α is over $LIST\ D$, and, as usual, $head\ (u : v) = u$ and $tail\ (u : v) = v$. It follows that

$$\begin{array}{lclll} HEAD\ (CONS\ \alpha\ \beta) & = & \alpha & & \text{if } \beta \neq FAIL \\ TAIL\ (CONS\ \alpha\ \beta) & = & \beta & & \text{if } \alpha \neq FAIL \end{array}$$

and so $HEAD$ and $TAIL$ take after their smaller brethren.

7 Examples

Applying the analysis method of the previous section to the definitions in Figures 1 and 5 gives the following results:

$$\begin{array}{lcl} length^1\ \alpha & = & NIL \sqcup CONS\ ABS\ (length^1\ \alpha) \\ before^1\ \alpha & = & NIL \sqcup CONS\ (STR\ \&\ HEAD\ \alpha)\ (ABS \sqcup before^1\ (TAIL\ \alpha)) \\ doubles^1\ \alpha & = & NIL \sqcup CONS\ (HEAD\ \alpha)\ (doubles^1\ (TAIL\ \alpha)) \\ append^1\ \alpha & = & NIL \sqcup CONS\ (HEAD\ \alpha)\ (append^1\ (TAIL\ \alpha)) \\ append^2\ \alpha & = & \alpha \sqcup append^2\ (TAIL\ \alpha) \\ reverse^1\ \alpha & = & NIL \sqcup CONS\ (HEAD\ (append^2\ \alpha))\ (rev^1\ (append^1\ \alpha)) \end{array}$$

$$
\begin{array}{lcl}
\mathit{append}\ \mathit{xs}\ \mathit{zs} & = & \textbf{case}\ \ \mathit{xs}\ \ \textbf{of} \\
 & & \quad [\,] \qquad \Rightarrow\ \ \mathit{zs} \\
 & & \quad y : \mathit{ys}\ \ \Rightarrow\ \ y : \mathit{append}\ \mathit{ys}\ \mathit{zs} \\
 & & \\
\mathit{reverse}\ \mathit{xs} & = & \textbf{case}\ \ \mathit{xs}\ \ \textbf{of} \\
 & & \quad [\,] \qquad \Rightarrow\ \ [\,] \\
 & & \quad y : \mathit{ys}\ \ \Rightarrow\ \ \mathit{append}\ (\mathit{revese}\ \mathit{ys})\ [y]
\end{array}
$$

Figure 5: More example programs

$$
\begin{array}{lcl}
\mathit{append}^1\ \alpha & = & (\mathit{xs}^{\mathit{xs}}\ \mathit{NIL}\ \&\ \mathit{zs}^{\mathit{xs}}\ \alpha) \\
 & & \sqcup(\mathit{xs}^{\mathit{xs}}\ (\mathit{CONS}\ ((y : \mathit{append}\ \mathit{ys}\ \mathit{zs})^{y}\ \alpha) \\
 & & \qquad\qquad\qquad ((y : \mathit{append}\ \mathit{ys}\ \mathit{zs})^{\mathit{ys}}\ \alpha)) \\
 & & \quad \&(y : \mathit{append}\ \mathit{ys}\ \mathit{zs})^{\mathit{xs}}\ \alpha)) \\
 & = & (\mathit{NIL}\ \&\ \mathit{ABS}) \\
 & & \sqcup((\mathit{CONS}\ (y^{y}\ (\mathit{HEAD}\ \alpha)\ \&\ (\mathit{append}\ \mathit{ys}\ \mathit{zs})^{y}\ (\mathit{TAIL}\ \alpha)) \\
 & & \qquad\qquad (y^{\mathit{ys}}\ (\mathit{HEAD}\ \alpha)\ \&\ (\mathit{append}\ \mathit{ys}\ \mathit{zs})^{\mathit{ys}}\ (\mathit{TAIL}\ \alpha))) \\
 & & \quad \&\mathit{ABS}) \\
 & = & \mathit{NIL}\ \&\ (\mathit{CONS}\ (\mathit{HEAD}\ \alpha)\ (\mathit{append}^1\ (\mathit{TAIL}\ \alpha))) \\
 & & \\
\mathit{append}^2\ \alpha & = & (\mathit{xs}^{\mathit{zs}}\ \mathit{NIL}\ \&\ \mathit{zs}^{\mathit{zs}}\ \alpha) \\
 & & \sqcup(\mathit{xs}^{\mathit{zs}}\ (\mathit{CONS}\ ((y : \mathit{append}\ \mathit{ys}\ \mathit{zs})^{y}\ \alpha) \\
 & & \qquad\qquad\qquad ((y : \mathit{append}\ \mathit{ys}\ \mathit{zs})^{\mathit{ys}}\ \alpha)) \\
 & & \quad \&(y : \mathit{append}\ \mathit{ys}\ \mathit{zs})^{\mathit{zs}}\ \alpha)) \\
 & = & (\mathit{ABS}\ \&\ \alpha) \\
 & & \sqcup(\mathit{ABS} \\
 & & \quad \&(y^{\mathit{zs}}\ (\mathit{HEAD}\ \alpha)\ \&\ (\mathit{append}\ \mathit{ys}\ \mathit{zs})^{\mathit{zs}}\ (\mathit{TAIL}\ \alpha))) \\
 & = & \alpha \sqcup \mathit{append}^2\ (\mathit{TAIL}\ \alpha)
\end{array}
$$

Figure 6: Context analysis of *append*

As usual, this assumes α is strict and not FAIL; otherwise, we use the rule $f^i\ \alpha = \alpha \rhd f^i\ \alpha'$. For the interested reader, details of the derivation for *append* are shown in Figure 6. The results have been simplified to improve readability, by reducing terms of the form $\mathit{ABS}\ \&\ \beta$ to β.

Not surprisingly, recursive function definitions yield recursive definitions of projection transformers. Using the finite domains of Sections 4 and 5, we can solve these in the usual way by taking the limits of ascending Kleene chains [CP85]. For instance, for *length* we define:

$$\begin{array}{lcl} length^{1(0)}\ \alpha & = & FAIL \\ length^{1(i+1)}\ \alpha & = & NIL \sqcup CONS\ ABS\ (length^{1(i)}\ \alpha) \end{array}$$

We then have

$$\begin{array}{lcl} length^{1(0)}\ STR & = & FAIL \\ length^{1(1)}\ STR & = & NIL \sqcup CONS\ ABS\ FAIL \\ & = & FIN\ FAIL \\ length^{1(2)}\ STR & = & NIL \sqcup CONS\ ABS\ (FIN\ FAIL) \\ & = & FIN\ ABS \\ length^{1(3)}\ STR & = & NIL \sqcup CONS\ ABS\ (FIN\ ABS) \\ & = & FIN\ ABS \end{array}$$

and so $length^1\ STR = FIN\ ABS$, just as we expected: *length* must be given a finite list, but ignores the lists elements.

Because the domains are finite, the ascending chains are guaranteed to reach a fixpoint after a finite number of iterations. Further, *no algebraic simplification methods are required.* For any given list domain, we may construct finite tables for calculating the relevant functions ($\sqcup$, &, *CONS*, *HEAD*, *TAIL*) and then compute the limits as above in a completely mechanical fashion.

Applying the above techniques we can derive, among other results, the following:

$$\begin{array}{lllcl} (1) & & before^1\ ID & = & ABS \sqcup INF\ STR \\ (2) & & before^1\ STR & = & INF\ STR \\ (3) & & doubles^1\ STR & = & STR \\ (4) & & doubles^1\ (INF\ STR) & = & INF\ STR \\ (5) & & append^1\ (FIN\ STR) & = & FIN\ STR \\ (6) & & append^2\ (FIN\ STR) & = & FIN\ STR \\ (7) & & append^1\ (INF\ STR) & = & INF\ STR \\ (8) & & append^2\ (INF\ STR) & = & ABS \sqcup INF\ STR \\ (9) & & reverse^1\ STR & = & FIN\ ID \\ (10) & & reverse^1\ (FIN\ STR) & = & FIN\ STR \end{array}$$

Line (1) shows that *before* is head strict (recalling that $H = ABS \sqcup INF\ STR$), and line (2) shows in addition that it is strict. (In fact, line (1) follows immediately from line (2), by an application of the guard rule.) Line (3) shows that *doubles* is strict, but says nothing else, while line (4) shows that *doubles* is head strict in a head strict context. Lines (5–6) show that in a context requiring a completely evaluated list, *append* must completely evaluate both its arguments. Lines (7–8) show that in a head strict context *append* is head strict in both arguments, but it is only strict in the first argument. Line (9) shows that the argument to reverse must be a finite list, and line (10) shows that if *reverse* is evaluated in a head and tail strict context, then so is its argument.

The results for *reverse* are particularly significant, since in the original work on contexts [Hug85] the analysis of *reverse* was more problematic. The analysis method has also been applied to a few other functions (the other common definition of reverse, insertion sort) with equally good results.

What are the method's limitations? The major one is that conditional and case expressions have a special role. For instance, if we define a function *cond* by

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

and then replace an arbitrary conditional by an equivalent call on *cond*, then analysis of the transformed function may give a much worse result. This is worrying, although more research is needed to discover whether this will be a significant problem in practice.

8 Relation to other work

Burn's evaluation transformers. Geoffrey Burn has suggested evaluation transformers as a way of controlling parallelism in a functional language implementation [Bur87]. There are some close relationships between his work and ours, but also some important differences.

Burn introduces four evaluators, $\mathcal{E}_0, \mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3$, which correspond to the four projections ID, STR, T', $H' \sqcap T'$. His main result can be re-phrased in our terms as follows: it is safe to use an evaluator $\mathcal{E}_i$ (in the sense that this will waste no work) whenever it is safe to apply the corresponding projection (in the sense that this will not change the result).

For example, writing $f\ (\mathcal{E}_2\ e)$ means that the spine of e can be evaluated *in parallel* with the application of f. This is rather different than writing $f\ (\ T'\ e)$, which insists that the entire spine of e is evaluated *before* it is passed to f. However, $f\ (\mathcal{E}_2\ e)$ wastes no work (evaluates no unneeded portion of the spine) only when $f\ (\ T'\ e) = f\ e$ (that is, only when f is tail-strict). The key to Burn's work is this link between an operational notion of safety ("wastes no work") and a denotational one ("doesn't change the result").

Burn's analysis is based on Wadler's previous work [Wad87]. Whether evaluation transformers can take advantage of the extra information revealed by projection analysis (such as absence or head strictness) is one of many remaining open questions.

Dybjer's inverse image analysis. Inspired by earlier work on context analysis, Peter Dybjer has devised a method of analysis based on inverse images of open sets [Dyb87]. The method has a simple and elegant mathematical foundation. Like earlier work on contexts, it uses algebraic manipulation to solve equations.

Interestingly, the open sets used by Dybjer correspond exactly to a restricted class of projections. Namely, open sets correspond to projections α with the additional restriction that for each u, either $\alpha\ u = \lightning$ or $\alpha\ u = u$. It is easy to see that with this restriction, the & operation defined in Section 6.4 is exactly equivalent to $\sqcap$. The projections STR and T' satisfy this restriction, while ABS and H' do not. Thus open sets can describe strictness and tail strictness, but appear ill-suited for describing absence and head strictness.

Hall's strictness patterns. Cordelia Hall's strictness analyser is based on strictness patterns [HW87]. There are striking similarities between strictness patterns and projections; compare the strictness pattern laws, $\$\$\pi = \$\pi$ and $\$\pi \sqsupseteq \pi$, to the projection laws,

$\alpha\ \alpha\ u = \alpha\ u$, and $\alpha\ u \sqsubseteq u$ (note that strictness patterns reverse the ordering). However, there appear to be no strictness pattern corresponding to ABS.

Unlike us, Hall can extract useful strictness from a list in which, say, every other element is strict. Also unlike us, Hall has examined the question of how to generate different versions of a procedure depending on the context in which it is called. Overall, the two works appear to be complementary. Whereas our work has stressed simple foundations, Hall's has stressed practical issues in building a prototype.

Wray's strictness analyser. The relation between the four point domain $\{ID, STR, ABS, FAIL\}$ and the work of Stuart Wray [Wra86, FW86] has already been mentioned. Wray's analyser also handles higher-order functions and a flexible type system; this inspired the similar extensions for backwards analysis outlined in [Hug87b].

Finally, here are two applications of projections outside of strictness analysis.

Bjerner's complexity analysis. An important open problem is analysis of the time and space complexity of lazy functional programs. Bror Bjerner has devised an elegant solution to this problem for the programming language of Martin-Löf's type theory [Bje87]. The solution makes use of *evaluation notes* to describe how much of the result of a program is required. It appears straightforward to adapt projections for use as evaluation notes, and to adapt Bjerner's method to lazy functional languages.

Pre-order traversal. In connection with Wadler's work on the listless transformer, it was necessary to describe the notion of pre-order traversal of a data structure [Wad85b]. This was done by introducing a function PRE, satisfying $PRE \sqsubseteq ID$ and $PRE \circ PRE = PRE$. A function f was pre-order if $PRE \circ f = PRE \circ f \circ PRE$. In other words, pre-order traversal was characterized by a projection. (Thus, this paper can be applied directly to solving the problem, posed by that paper, of how to analyse pre-order traveral.)

It was exactly at this time that the two authors were pursuing the work on strictness analysis described in the introduction. But it was not until a year later that Wadler realized that this approach could describe the work of Hughes!

9 Conclusion

This work has provided a simpler explanation of contexts than available previously. The finite domains for lists presented here make it possible to solve recursive equations in a straightforward way, using standard fixpoint techniques, which has advantages over the previous method of algebraic manipulation.

An important next step is to extend backwards analysis to languages with higher-order functions and polymorphic typing; a way of doing this is outlined in [Hug87b]. Some applications of projections outside strictness analysis were suggested in the last section. We hope the approach presented here will provide fertile soil for future developments.

Our projection for strictness analysis is a rosy future.

Acknowledgements. This work owes a debt to the Programming Methodology Group at Chalmers University of Technology, Göteborg. The first paper on context analysis was written while Hughes was a visitor at Chalmers, and this paper was written while Wadler was a visitor there. We are grateful to the members and staff of the PMG for the enjoyable, stimulating, and supportive environment they provided.

We thank Richard Bird, Geoffrey Burn, Peter Dybjer, Cordelia Hall, Thomas Johnsson, Kent Karlsson, Jon Fairbairn, Simon Peyton-Jones, Staffan Truvè, and Stuart Wray for fruitful discussions and for comments on an earlier draft of this paper. Wadler is particularly grateful to Peter Dybjer for comments on the mathematical rigour (or lack of it) in an earlier version of this work.

Part of this work was performed while Wadler was on a research fellowship supported by ICL.

References

[Abr85] S. Abramsky. Strictness analysis and polymorphic invariance. In N. Jones and H. Ganzinger, editors, *Workshop on Programs as Data Objects*, Springer-Verlag, Copenhagen, October 1985. LNCS 217.

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

[Aug84] L. Augustsson. A compiler for lazy ML. In *ACM Symposium on Lisp and Functional Programming*, pages 218–227, Austin, 1984.

[BHA85] G. L. Burn, C. L. Hankin, and S. Abramsky. The theory of strictness analysis for higher-order functions. In N. Jones and H. Ganzinger, editors, *Workshop on Programs as Data Objects*, Springer-Verlag, Copenhagen, October 1985. LNCS 217.

[Bje87] B. Bjerner. Complexity analysis of programs in type theory. Programming Methodology Group, Chalmers University of Technology, Göteborg, Sweden, 1987.

[Bur87] G. L. Burn. Evaluation transformers—a model for the parallel evaluation of functional languages. In *Conference on Functional Programming Languages and Computer Architecture*, Portland, Oregon, September 1987.

[CP85] C. Clack and S. L. Peyton-Jones. Strictness analysis—a practical approach. In *Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, 1985.

[Dyb87] P. Dybjer. Computing inverse images. In *International Conference on Automata, Languages, and Programming*, 1987.

[FW86] J. Fairbairn and S. C. Wray. Code generation techniques for functional languages. In *ACM Symposium on Lisp and Functional Programming*, pages 94–104, Boston, 1986.

[HB85] P. Hudak and A. Bloss. The aggregate update problem in functional programming systems. In *12'th ACM Symposium on Principles of Programming Languages*, pages 300–314, January 1985.

[Hug84] R. J. M. Hughes. *Why functional programming matters.* Technical Report, Programming Methodology Group, Chalmers University of Technology, Göteborg, Sweden, 1984.

[Hug85] R. J. M. Hughes. Strictness detection in non-flat domains. In N. Jones and H. Ganzinger, editors, *Workshop on Programs as Data Objects*, Springer-Verlag, Copenhagen, October 1985. LNCS 217.

[Hug87a] R. J. M. Hughes. Analysing strictness by abstract interpretation of continuations. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987, to appear.

[Hug87b] R. J. M. Hughes. Backwards analysis of functional programs. University of Glasgow research report CSC/87/R3, March 1987.

[HW87] C. V. Hall and D. S. Wise. Compiling strictness into streams. In *14'th ACM Symposium on Principles of Programming Languages*, pages 132–143, Munich, January 1987.

[HY85] P. Hudak and J. Young. Higher order strictness analysis in untyped lambda calculus. In *12'th ACM Symposium on Principles of Programming Languages*, pages 97–109, January 1985.

[Joh84] T. Johnsson. Efficient compilation of lazy evaluation. In *ACM Symposium on Compiler Construction*, 1984.

[Joh87] T. Johnsson. Attribute grammars as a paradigm for functional programming. Programming Methodology Group, Chalmers University of Technology, Göteborg, Sweden, 1987.

[Myc81] A. Mycroft. *Abstract interpretation and optimising transformations for applicative programs.* PhD thesis, University of Edinburgh, 1981.

[PC87] S. L. Peyton-Jones and C. Clack. Finding fixpoints in abstract interpretation. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987.

[Pey87] S. L. Peyton-Jones. *Implementing Functional Languages using Graph Reduction.* Prentice-Hall, 1987.

[Sch86] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Newton, Massachusetts, 1986.

[Sco81] D. S. Scott. *Lectures on a mathematical theory of computation.* Technical Report PRG-19, Oxford University Programming Reseach Group, May 1981.

[Sco82] D. S. Scott. Domains for denotational semantics. In *Conference on Automata, Languages and Programming*, pages 577–613, Springer-Verlag, July 1982. LNCS 140.

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

[Wad85a] P. L. Wadler. How to replace failure by a list of successes. In *Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, 1985.

[Wad85b] P. L. Wadler. Listlessness is better than laziness II: composing listless functions. In N. Jones and H. Ganzinger, editors, *Workshop on Programs as Data Objects*, Springer-Verlag, Copenhagen, October 1985. LNCS 217.

[Wad87] P. L. Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987, to appear.

[Wra86] S. C. Wray. *Implementation and Programming Techniques for Functional Languages.* PhD thesis, University of Cambridge, January 1986.

[YH86] J. Young and P. Hudak. *Finding fixpoints on function spaces.* Technical Report YALEU/DCS/RR-505, Yale University Dept. of Computer Science, December 1986.