Evaluation Transformers - A Model for the Parallel Evaluation of Functional Languages (Extended Abstract) *

Geoffrey L. Burn

GEC Research Ltd
Hirst Research Centre
East Lane
Wembley
Middx. HA9 7PP
United Kingdom

Electronic mail : ...uk.ac.ukc!uk.co.gec-rl-hrc!geoff

ABSTRACT

If we are not careful, a parallel machine may become swamped with the computation of the expressions whose values are not needed in order to produce the result from a program. We give a semantic criterion which ensures that this does not happen.

An abstract interpretation can be developed which gives the definedness of a function in terms of the definedness of its arguments. Traditionally this has been used to give a *strictness analysis* which has been interpreted to say how much the argument in an application can be evaluated in parallel with the application while still satisfying the semantic criterion. Strictness analysis however only takes into account local information.

By taking into account contextual information of an application, we are able to find that in many cases more evaluation of the argument is allowed. This information is available from the same abstract interpretation that is used for strictness analysis. Given how much evaluation is allowed of a function application, an *evaluation transformer* says how much evaluation of the argument in the application is allowed. This leads to a natural model of the parallel evaluation of functional languages.

Keywords : Functional Languages, Parallel Reduction, Abstract Interpretation, Strictness Analysis, Evaluation Transformers

* Research partially funded by ESPRIT Project 415 : Parallel Architectures and Languages for AIP - A VLSI-Directed Approach.

1. Parallelism in the Evaluation of Functional Programs.

There are two broad classes of ways we may choose to obtain parallelism in the evaluation of functional languages which have no explicit parallel constructs. A machine may employ *speculative parallel evaluation*, where all possible redexes in a graph are reduced in parallel, or it may use *conservative parallel evaluation*, where it only evaluates an expression if it knows it will need its value.

Speculative parallelism wastes machine resources by evaluating expressions which may eventually be discarded. For example, in the expression

if condition then e_1 *else* e_2

the value of only one of e_1 and e_2 will be needed, depending on the truth of the *condition*. The problem is compounded in languages which allow the writing of expressions denoting infinite computations, for such computations may try and consume infinite amounts of resources.(*) This would imply the need to garbage collect infinite processes, which is a difficult problem we would like to avoid.

Therefore, we have investigated a conservative parallel reduction model. What is required is a method for finding out which redexes in a program graph are going to be eventually reduced in the evaluation of a program.

By only ever evaluating the left-most outer-most redex and evaluating expressions only as far as head normal form, lazy evaluation ensures that no expression is evaluated more than is needed to produce the result of a calculation. While this is perfectly satisfactory for a sequential machine, it is hardly useful for a parallel machine for it only ever allows one expression to be evaluated at a time. The problem is that lazy evaluation is overly pessimistic about which expressions are going to be needed - it only knows that the left-most outer-most redex is needed.

Another way of looking at lazy evaluation is to notice that it never initiates a non-terminating computation unless the semantics of the original expression to be evaluated was undefined, that is, bottom. Our problem then reduces to ensuring that we do not initiate a computation unless its non-termination would imply that the semantics of the overall expression was bottom. We will call this our *semantic criterion*. Note that this precludes the evaluation of expressions whose values are not needed, even if it could be determined that their evaluation would terminate. A *safe* evaluation strategy is one which obeys the semantic criterion.

(*) An infinite computation does not necessarily mean no result is produced. When one has structured data types, a computation may produce a finite or unbounded amount of output as well as proceeding forever.

Our semantic criterion seems to be a natural way to specify the behaviour of a parallel evaluation strategy which does not evaluate an expression unless it its value is needed at some time in the computation. This implies that there is no wasted work, and also means that there is no need to garbage collect infinite processes unless the whole program is undefined. Note how the semantic criterion relates strongly the operational behaviour of an evaluation strategy and the denotational semantics of a program. Abstract interpretation will be used to gain the information necessary for checking the semantic criterion.

By giving a different interpretation, an *abstract interpretation*, to the symbols in a programming language, we are sometimes able to find out information about a program without running it. An abstract interpretation for determining the definedness of functions in terms of the definedness of their arguments has been developed in a series of papers. Mycroft [Mycroft 1981] developed a strictness analysis for first-order functions over atomic data types(*). This was extended in [Burn, Hankin and Abramsky 1986] to a strictness analysis for higher-order functions. The work depends on programs being monotyped, but see [Abramsky 1985a]. Wadler [Wadler 1987] introduced an abstract domain for structured data types such as lists. All of these various strands were drawn together in [Burn 1987a] where a framework for the abstract interpretation of functional languages was developed and applied to this problem. A related analysis is also developed in [Hudak and Young 1985], based on giving a different style of semantics to a language. It is designed for typeless languages.

Traditionally this abstract interpretation has been used to give a *strictness analysis* of functional programs, that is, finding if a function application is undefined when one of its arguments is undefined. However, this loses information, for it only tests the semantic criterion locally. By looking at the same abstract interpretation in another way, we are able to determine more parallelism information, which leads to a natural model for the parallel evaluation of functional languages. We call the results of this analysis *evaluation transformers*, for they tell how much evaluation can be done to an argument of a function given that we can do a certain amount of evaluation of the function application.

This paper does not develop a new abstract interpretation for finding out definedness information about functions. Rather it gives an intuitive introduction to evaluation transformers, showing how they give more information than strictness analysis, the way

(*) An *atomic data type* is one which has a flat domain as its usual interpretation. Integers and booleans are two examples.

to determine them from the abstract interpretation developed in the abovementioned series of papers, and how they give rise to a parallel evaluation model for functional languages which obeys the semantic criterion and is as natural as lazy evaluation is a natural model for the sequential implementations of functional languages. Therefore, no proofs are given in the paper; a full mathematical treatment will be the subject of a later paper, or can be found in [Burn 1987a].

It is important to note that this work rests on the foundations of an abstract interpretation which deals with higher-order functions.

This paper is organised as follows. The concept of an evaluator is covered in section 2, and section 3 introduces the abstract interpretation of several functions which are used as examples in sections 4 and 5, which discuss respectively strictness analysis and evaluation transformers. Section 6 shows how evaluation transformers lead to a natural parallel reduction model. In section 7 we lift the restrictions that have been made in order to simplify the exposition in the main body of the text, and in section 8 we compare our work with [Hughes 1985] and [Wadler and Hughes 1987].

We will write our examples in the language Miranda(*) [Turner 1985].

2. "Evaluation" of Expressions.

So far we have said that we wish to "evaluate" an expression without being specific as to how much evaluation can be done. Implementations of functional languages normally keep reducing an expression until it is in *head normal form* (**). A λ-expression is in *head normal form* if and only if it is of the form

$$\lambda x_1.\cdots\lambda x_n.(v\ M_1\ \cdots\ M_m)$$

where

(i) $n, m \geq 0$,

(ii) v is a variable (i.e. x_i for some i) or a constant, and

(*) Miranda is a trademark of Research Software Ltd.

(**) Functional language evaluators often evaluate functions only as far as *weak head normal form*, which differs from head normal form only in the way that it treats functions [Peyton Jones 1987]. An expression which stands for a function is in weak head normal form when there is a λ at the top-level. By only evaluating expressions to weak head normal form we remove the problem of having to rename variables within expressions, making implementation a lot easier [Peyton Jones 1987]. Consider the expression

$$\lambda x.((\lambda y.\lambda x.y)x)$$

which is weak head normal form but not head normal form. To reduce this to head normal form requires a renaming of one of the bound variables to obtain $\lambda x.\lambda z.x$.

(iii) if v is a constant, then m is less than the arity of v.

For atomic data types such as integers and booleans, that is types which have a flat domain as their normal interpretation, this is the maximum amount of evaluation that can be performed. For lists, an expression is in head normal form when it has been reduced to *nil* or a *cons* of two unevaluated expressions. We know however, that in some function applications the argument will need more reduction than just to head normal form. The function

length [] = 0
length (*x*:*xs*) = 1 + *length xs*

will eventually need to traverse the whole of the argument list, but will not need any of the values of elements of the list. The function

sumlist [] = 0
sumlist (*x*:*xs*) = *x* + *sumlist xs*

needs to traverse the whole of its argument list and also obtain the values of the elements of the list. We will call the process of recursively evaluating the second argument of *cons* "creating the *structure*" of the list. (This process terminates if *nil* is reached, that is, the list is finite.)

There is a similar idea for all recursively defined types, such as numeric binary trees which have the Miranda type definition :

tree ::= *NIL_TREE* | *NODE num tree tree*

where the second and third arguments to the *NODE* constructor are recursively evaluated. Essentially, evaluating the structure of an expression is unfolding the recursive part of the data type definition.

We will say that we can evaluate an expression using a particular *evaluator*, and call an evaluator which evaluates expressions to head normal form ξ_1, an evaluator which evaluates the structure of a list ξ_2, and an evaluator which evaluates the structure of a list and every element of the list to head normal form ξ_3. For completeness, the evaluator ξ_0 does no evaluation. In section 4 we will see that any argument to *length* can indeed be evaluated using ξ_2 and that any argument to *sumlist* can be evaluated using ξ_3, as our intuition above suggested.

Note that an implementation does not need to create the whole structure of a list if it knows that it is safe to do so. This is because the information we will find out says that we may use a particular evaluator, not that we must use it. (It also says that we may not use a stronger evaluator.) If the list was very long, then creating its structure

may cause space problems in a machine. Therefore, it may be implemented as an evaluator which created the structure of the list in small chunks, which is activated by a consumer requiring some more of the list. ξ_3 can be implemented in a similar manner. Also, if suitable mechanisms are incorporated into an architecture (see for example [Bevan et al 1987]), each *cons* cell of a list being evaluated by ξ_2 or ξ_3 can be made available when it is created. This means that the whole list does not have to be created before it can be consumed.

We chose such evaluators because they treat each element of a list in a uniform way. When the structure is more complex than a list of atomic data objects, then we may wish to add more evaluators. This causes no further problems, but would clutter our presentation; we return briefly to this in section 7.

3. Abstract Interpretation.

In this section we will define some functions and give their definedness abstract interpretation. We assume the reader is familiar with this particular interpretation [Burn, Hankin and Abramsky 1986], [Wadler 1987], [Burn 1987a]. The abstract domain used for atomic base types, from [Mycroft 1981], has two values :

0 - representing the undefined value
1 - representing any value

and is often called 2, for it is the two point domain. For list types the domain of [Wadler 1987] is used :

0 - representing the undefined list
1 - representing infinite lists and lists which have an undefined tail after a finite number of elements, including the undefined list
2 - representing all those represented by 1, plus all finite lists with at least one undefined element
3 - representing all lists

We need to introduce some notation for our discussion. It σ is some type, by D_σ we will mean the abstract domain for the type σ, $\top_{D_\sigma}$ is the top element of the abstract domain D_σ (e.g. 1 for atomic base types and 3 for lists) and $\bot_{D_\sigma}$ is the bottom element of the abstract domain D_σ. For the function type $\sigma \to \tau$, the abstract domain is $D_\sigma \to D_\tau$, that is, the set of continuous functions from D_σ to D_τ. When using the tests given in Facts 4-1 and 5-1 of this paper, we will only need the top and bottom elements of the abstract domains for the function spaces. For the type $\sigma_1 \to \cdots \to \sigma_n \to \tau$, the top element is $\lambda x^{D_{\sigma_1}}. \cdots \lambda x^{D_{\sigma_n}}.\top_{D_\tau}$ and the bottom element is is $\lambda x^{D_{\sigma_1}}. \cdots \lambda x^{D_{\sigma_n}}.\bot_{D_\tau}$ As with the

cases of atomic base domains and lists, the bottom element of each of the abstract domains represents only the bottom element of the corresponding real function space and the top element represents all elements in that function space.

These abstract domains are useful because they capture the way that the evaluators we have chosen behave. For example, the evaluator ξ_1 will fail to terminate if and only if the expression being evaluated denotes an undefined value, that is, the value represented by the bottom of the appropriate abstract domain (0 for atomic base types and lists). In a similar manner, because ξ_2 evaluates the structure of a list, it will fail to terminate precisely in the case that the list is not finite, that is if an expression denotes any value represented by 1. Finally, ξ_3 will also fail to terminate if the expression denotes a list which is either not finite or has any undefined elements in it, that is, it fails to terminate on any expression denoting an element represented by 2. For this reason, we will say that the evaluator ξ_1 *corresponds* to the bottom of the appropriate abstract domain, that ξ_2 *corresponds* to 1 and that ξ_3 *corresponds* to 2.

We give three more function definitions, and then give the abstract interpretations of all five examples plus those for *head* and *tail*. Note that *map* is a higher-order function.

append [] *ys* = *ys*
append (*x*:*xs*) *ys* = *x*:(*append xs ys*)

reverse [] = []
reverse (*x*:*xs*) = *append* (*reverse xs*) (*x*:[])

map f [] = []
map f x:*xs* = (*f x*):(*map f xs*)

In the following we will denote the abstract interpretation of a function by placing a bar over the name of the function. Thus the abstract interpretation of f is written $\bar{f}$. [Wadler 1987] and [Burn 1987a] show how to derive the abstract interpretation of a function. Here we content ourselves with giving the results for our examples. Table 3-1 gives the abstract interpretation of *sumlist*, *length* and *head*. The abstract interpretation of *reverse* and *tl* are given in Table 3-2. Table 3-3 gives the values of $\overline{append}\ \overline{xs}\ \overline{ys}$. Remembering that we are working in a (mono-) typed framework, Table 3-4 gives the values of $\overline{map}\ \bar{f}\ \overline{xs}$ for a *map* of type $(int \rightarrow int) \rightarrow list\ int \rightarrow list\ int$.

It can seen that these abstract interpretations capture our intuitions about the definedness of functions. For example, the function *sumlist* is undefined (has value 0 in the abstract domain) whenever the argument is not a finite list with no undefined elements in it (has the value 2 in the abstract domain). The only slightly curious result is the value of $\overline{map}\ (\lambda x^2.0)\ 3$ which is 3 rather than 2 (recalling that $\lambda x^2.0$ represents the

Table 3-1.
Abstract Interpretation of *sumlist*, *length*, and *hd*.

$\overline{xs}$	$\overline{sumlist}(\overline{xs})$	$\overline{length}(\overline{xs})$	$\overline{hd}(\overline{xs})$
0	0	0	0
1	0	0	1
2	0	1	1
3	1	1	1

Table 3-2.
Abstract Interpretation of *reverse* and *tl*.

$\overline{xs}$	$\overline{reverse}(\overline{xs})$	$\overline{tl}(\overline{xs})$
0	0	0
1	0	1
2	2	3
3	3	3

Table 3-3.
Abstract Interpretation of *append*.

$\overline{ys}$ \ $\overline{xs}$	0	1	2	3
0	0	1	1	1
1	0	1	1	1
2	0	1	2	2
3	0	1	2	3

bottom function and applying the bottom function to each element of a finite list returns a finite list all of whose elements are bottom). The value 3 is correct because when *map* is applied to the empty list, which has 3 as its abstract interpretation (as it is a finite list with no bottom elements in it!), then it returns the empty list. As the abstract interpretation has to capture all possible results of an application of *map*, then the result in this case has to be 3.

Table 3-4.
Abstract Interpretation of *map*.

$\overline{xs}$ \ $\bar{f}$	$\lambda\bar{x}^2.0$	$\lambda\bar{x}^2.x$	$\lambda\bar{x}^2.1$
0	0	0	0
1	1	1	1
2	2	2	3
3	3	3	3

We will now show how to use this information to control the parallel evaluation of expressions.

4. Strictness Analysis.

Suppose that we know we have to evaluate a function application. Because we have to evaluate the application, we know by the semantic criterion that if the application is undefined, that is, evaluating the application will cause a non-terminating computation, then the original expression must have been undefined. If we therefore choose an evaluator for the argument so that it does as much evaluation as possible, but does not initiate a non-terminating computation unless the function application is undefined, then we have preserved the semantic criterion.

We are thus interested in finding out when a function is undefined. Specifically, we want to find the maximally defined argument for which an application of a function to that argument is still undefined. This has been called *strictness analysis* in the literature. The following fact, proved formally as Theorem 3.5.1 in [Burn 1987a], allows us to say when arguments to functions may be evaluated in parallel.

Fact 4-1:

Suppose that f is a function of type $\sigma_1 \rightarrow \cdots \sigma_n \rightarrow \tau$, and that $\bar{f}$ is the abstract interpretation of f. If

$$\bar{f} \top_{D_{\sigma_1}} \cdots \top_{D_{\sigma_{i-1}}} \overline{s_i} \top_{D_{\sigma_{i+1}}} \cdots \top_{D_{\sigma_n}} = \bot_{D_\tau}$$

then in any application of f the semantic criterion allows the evaluation of the ith argument to f using the evaluator corresponding to $\overline{s_i}$.

□

The top elements used in the test are capturing the idea of "for any possible value of the argument" (c.f. that the point 3 stood for any list and the point 1 for any atomic base domain value). This fact allows us to choose any $\overline{s_i}$ satisfying the condition, but clearly we will choose the largest one because this will allow us to use the most powerful evaluator for the argument.

Three examples will illustrate the use of the test. The function *length* takes one argument. From Table 3-1 we find that

$$\overline{length}\ 0 = \overline{length}\ 1 = 0.$$

Since ξ_1 corresponds to 0 and ξ_2 corresponds to 1, we are allowed by the Theorem 4-1 to evaluate the argument to *length* using either ξ_1 or ξ_2. Choosing the maximum value means we are allowed to do the maximum evaluation. This corresponds to our intuition that *length* needs to have a finite list as its argument or the application would be undefined.

Append is a function of two arguments. From Table 3-3 we find that the maximum $\bar{s}$ such that

$$\overline{append}\ \bar{s}\ 3 = 0$$

is 0, which corresponds to ξ_1, and so by the above fact we can evaluate the first argument to *append* in any application to head normal form. There is no $\bar{s}$ such that

$$\overline{append}\ 3\ \bar{s} = 0$$

and so we are not able to do any evaluation of the second argument.

Recalling that $\lambda x^2.1$ is the top element of the abstract domain for functions of type $int \rightarrow int$, from Table 3-4 we find that the maximum $\bar{s}$ such that

$$\overline{map}\ (\lambda x^2.1)\ \bar{s} = 0$$

is 0, which corresponds to ξ_1, and so the second argument to *map* in any application can be evaluated to head normal form. As the only sensible amount of evaluation for a function is ξ_1, we need only try the bottom of the appropriate type for the $\overline{s_i}$ in the test when the ith argument is a function. From Table 3-4 we see that

$$\overline{map}\ (\lambda x^2.0)\ 1 = 1 \neq 0$$

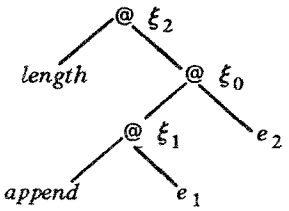
and so no evaluation is allowed on the functional argument to *map*.

We can use this information to label the function applications in a program with

evaluators which say how much evaluation can be made of an argument in an applications. Using the above information, in the application

length (*append* e_1 e_2)

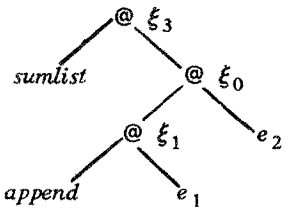
we can label the apply nodes as in the diagram :



This says that the expression *append* e_1 e_2 can be evaluated using ξ_2, the expression e_1 can be evaluated using ξ_1, and no evaluation is permitted on the expression e_2. Similarly, we can label the application

sumlist (*append* e_1 e_2)

as in the diagram :



which says that the argument to *sumlist* can be evaluated using ξ_3, while the first argument to *append* can be evaluated using ξ_1 and no evaluation is allowed of the second argument.

5. Evaluation Transformers

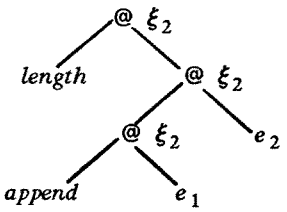
Strictness analysis is deficient because it does not take into account the contextual information about a function application.

Let us reconsider the expression

length (*append* e_1 e_2)

which was discussed in the previous section. The application nodes in the application of *append* were labelled, making sure that we did not initiate a non-terminating computation unless the application of *append* was undefined. We noted that an application of *length* is

only defined if its argument is a finite list. Therefore the overall expression can only be defined if the result of the application of *append* is a finite list. We know that an application of *append* can only return a list which is finite if both of its argument lists are finite. In this context therefore, it is safe to evaluate both of the arguments to *append* using ξ_2, because we are guaranteed that the top-level application of *length* is undefined whenever evaluating the arguments to *append* using ξ_2 initiates a non-terminating computation. Hence we are able to label the applications as in the diagram :



using stronger evaluators on the arguments to *append* and thus allowing more parallel evaluation.

In a similar manner, we are able to label all applications in

sumlist (*append* e_1 e_2)

with ξ_3 because the only way that we can obtain a finite list with no undefined elements in it from an application of *append* is if both of the arguments to *append* are finite lists with no bottom elements in them.

So the first important observation is that by taking into account the context of an application, we may be allowed to use stronger evaluators on some of the arguments than we could determine using strictness analysis, while still obeying our semantic condition.

A second important point is that an expression may occur in many different contexts within a program. For example, the expression e defined by

$$e = \textit{append}\ e_1\ e_2$$

may appear in several contexts, such as *hd* e , *tl* e and *sumlist* e . In the first case, we would only be allowed to evaluate e using ξ_1, and so evaluate the first and second arguments to *append* using ξ_1 and ξ_0 respectively, while in the last case ξ_3 can be used for both arguments. We must therefore associate with each argument to a function a mapping that takes an evaluator which is safe in a particular context and gives an evaluator which is safe for the argument in that context. We call such a mapping an *evaluation transformer*. The compiler can label function applications with evaluation transformers. Given an application and an evaluator which is safe, concurrent task can be initiated to evaluate the

argument. The amount of evaluation permitted is given by applying the evaluation transformer on the application node to the evaluator of the application.

This work is related to some ideas of [Cousot and Cousot 1979] where they introduced the ideas of *forward flow analysis* and *backward flow analysis*. The traditional use of abstract interpretation corresponds to a forwards analysis - putting in values and seeing if bottom was the result. However, the view of evaluation transformers, seeing what constraints are put on the inputs of a function given some constraint on the output of the function, corresponds to the ideas of a backwards flow analysis.

Evaluation transformers can be determined using exactly the same abstract interpretation as was used in strictness analysis. Essentially it amounts to reading the interpretation backwards. We use the following fact, proved formally as Theorem 4.2.1.1 in [Burn 1987a], to determine evaluation transformers.

Fact 5-1:

Suppose that f is a function of type $\sigma_1 \rightarrow \cdots \sigma_n \rightarrow \tau$, and that $\bar{f}$ is the abstract interpretation of f. If

$$\bar{f} \top_{D_{\sigma_1}} \cdots \top_{D_{\sigma_{i-1}}} \bar{s_i} \top_{D_{\sigma_{i+1}}} \cdots \top_{D_{\sigma_n}} \leqslant \bar{t}$$

then when the evaluator corresponding to $\bar{t}$ is allowed by the semantic criterion to evaluate an application of f, the evaluator corresponding to $\bar{s_i}$ is allowed to be used for the evaluation of the ith argument to f.

□

By putting $\bar{t}$ into the right-hand side of the test, we are taking into account the contextual information. Strictness analysis only ever put $\bot_{D_{\sigma_\tau}}$ for this value, which corresponds to ξ_1, and so was only finding out how much evaluation could be done when evaluating the application to head normal form was safe.

An evaluation transformer must be determined for each argument to a function. For each possible evaluator of an application of a function, the transformer must give an evaluator.

The evaluator ξ_0 means that it is not safe to do any evaluation of an expression. If it is not safe to do any evaluation of a function application, then it is not safe to do any evaluation of an argument.(*) Thus all evaluation transformers will give ξ_0 at ξ_0.

(*) This is true by our definition of safety. However, if one was to have a looser definition of safety which just ensured that no non-terminating computation was initiated unless the semantics of the

Examples of determining the evaluation transformers for functions are now given.

The function *reverse* has a list as its result and therefore an application of *reverse* may appear in the context of being evaluated by any of the evaluators ξ_0 to ξ_3. From the above discussion the evaluation transformer will be ξ_0 at ξ_0. We will denote the evaluation transformer for the ith argument of a function f by F_i, that is the name of the function in upper case letters, subscripted by the argument number. This notation has been borrowed from [Hughes 1985]. Thus we have determined that

$$REVERSE_1(\xi_0) = \xi_0$$

We must determine the values for the other possible evaluators.

The evaluator ξ_1 corresponds to the point 0 and so we put in the value 0 on the right-hand side of the above test and determine the maximum $\bar{s}$ such that

$$\overline{reverse}\ \bar{s} \leqslant 0$$

From the abstract interpretation of *reverse* in Table 3-2 we find that the maximum value is 1 which corresponds to ξ_2, and so

$$REVERSE_1(\xi_1) = \xi_2$$

We can see that this is intuitively true by noting that ξ_1 is asking for an expression to be evaluated to head normal form and that *reverse* must traverse the entire list before it can create the first *cons* cell, so the list must be finite if it is to be able to give a result in head normal form.

Corresponding to the evaluator ξ_2 we have the point 1, and from the abstract interpretation we see the maximum point satisfying the condition is again 1, and thus the evaluation transformer is ξ_2 at ξ_2. Note that in this case we have that $\overline{reverse}\ 1$ is 0, which is strictly less defined than 1. This says that it is impossible for *reverse* to return a list which is partial or infinite.

Finally, the point 2 corresponds to the evaluator ξ_3, and the abstract interpretation says that the maximum $\bar{s}$ satisfying the criterion is 2, and so ξ_3 is a safe evaluator for the argument in this case. We interpret this as saying that for an application of *reverse* to return a finite list with no bottom elements, then it must be given a finite list with no

original expression was undefined, then the evaluation of such an expression may terminate and so it would be perfectly safe to evaluate it. However, this potentially wastes machine resources for we may not need the value of the expression. Moreover, our abstract interpretation is not able to give us this information. However, see [Mycroft and Nielson 1983] and [Abramsky 1985b].

bottom elements as an argument. The evaluation transformer for *reverse* is given in Table 5-2.

For functions such as *length* and *sumlist*, which have results which come from flat domains like integers and booleans, there is only ever one sensible evaluator for an application which does any work, namely ξ_1, and so we have only to determine the evaluation transformer at that point. In a similar manner, as ξ_1 is the only evaluator for functions, when determining the evaluation transformers for a function which returns a function as its result, we need only put the bottom of the appropriate abstract domain into the right-hand side of the test in Theorem 5-1.

As an example of the use of Theorem 5-1 for a function of more than one argument, we find the evaluation transformer for the first argument of *append* when a safe evaluator for the application is ξ_2. Using the above test, we find the maximum $\bar{s}$ such that

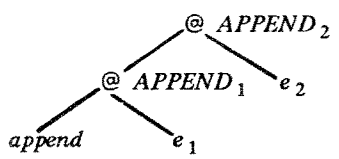
$$\overline{append}\ \bar{s}\ 3 \leqslant 1$$

noting that 3 is the top of the abstract domain for lists, and that 1 corresponds to the evaluator ξ_2. From the abstract interpretation of *append* given in Table 3-3, we find that the maximum value of $\bar{s}$ is 1, and thus

$$APPEND_1(\xi_2) = \xi_2.$$

The rest of the evaluation transformer for the first argument and the evaluation transformer for the second argument can be determined in a similar manner, and are given in Table 5-3.

Evaluation transformers are used to label arguments to functions and application nodes in the same way that we labelled them with evaluators in strictness analysis. It is important to note that labelling with evaluation transformers replaces labelling with evaluators, that is, we label with evaluation transformers and not evaluators. We can label the applications in *append* e_1 e_2 as in the diagram :



If, for example, the application was to be evaluated using ξ_2, say because we had applied *length* to it, then e_1 could be evaluated using $APPEND_1(\xi_2) = \xi_2$, and e_2 could be evaluated using $APPEND_2(\xi_2) = \xi_2$. At another time, ξ_1 may be a safe evaluator for the

application and so $\xi_1 = APPEND_1(\xi_1)$ is a safe evaluator for e_1, while $\xi_0 = APPEND_2(\xi_1)$ is a safe evaluator for e_2.

Table 5-1.
Evaluation Transformers for *sumlist*, *length* and *hd*.

E	$SUMLIST_1(E)$	$LENGTH_1(E)$	$HD_1(E)$
ξ_0	ξ_0	ξ_0	ξ_0
ξ_1	ξ_3	ξ_2	ξ_1

Table 5-2.
Evaluation Transformers for *reverse* and *tl*.

E	$REVERSE_1(E)$	$TL_1(E)$
ξ_0	ξ_0	ξ_0
ξ_1	ξ_2	ξ_1
ξ_2	ξ_2	ξ_2
ξ_3	ξ_3	ξ_2

Table 5-3.
Evaluation Transformers for *append*.

E	$APPEND_1(E)$	$APPEND_2(E)$
ξ_0	ξ_0	ξ_0
ξ_1	ξ_1	ξ_0
ξ_2	ξ_2	ξ_2
ξ_3	ξ_3	ξ_3

6. A Model for the Parallel Evaluation of Functional Languages.

Just as lazy evaluation is the natural model for the sequential evaluation of functional languages, programs annotated with evaluation transformers lead to a natural model for their parallel evaluation.

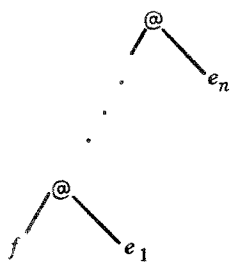
Table 5-4.
Evaluation Transformers for *map*.

E	$MAP_1(E)$	$MAP_2(E)$
ξ_0	ξ_0	ξ_0
ξ_1	ξ_0	ξ_1
ξ_2	ξ_0	ξ_2
ξ_3	ξ_0	ξ_2

One way of representing a functional program is as a graph of binary apply nodes. Thus the application

$$f\, e_1 \cdots e_n$$

would be represented in memory as :



Left-most outer-most reduction is obtained by traversing the spine of the application, until the function f is reached. A copy of the body of f is made, substituting pointers to the arguments e_1 to e_m if f needs m arguments, and the root of this graph overwrites the the application node which points to e_m. Traversal of the spine begins again at this node. An excellent coverage of graph reduction is given in [Peyton Jones 1987].

For parallel reduction we associate with each function its evaluation transformers. The function f has evaluation transformers F_1 to F_n. Suppose that the above expression is being evaluated using the evaluator ξ. Then the only thing that changes in the above evaluation mechansim is that when encountering the function f at the end of the spine, a task is initiated to evaluate each argument e_i with the evaluator $F_i(\xi)$. Of course, if $F_i(\xi)$ is ξ_0, then no parallel task is initiated.

There are some important things to note about this evaluation mechanism. Any particular expression is being evaluated using left-most outer-most reduction. The evaluation mechanism is not some sort of parallel-innermost reduction strategy, for in the

case that the f in the above example is a user-defined function, the evaluation of the expression e_i proceeds in parallel with the evaluation of the expression $f\, e_1 \cdots e_n$.

Evaluation transformers can be incorporated into a compiled implementation of functional languages. Most simply this can be done by having a case on the evaluator at the entry point of the code for a function. The code for each case initiates the evaluation of the argument expressions for which the evaluation transformers give a non-ξ_0 evaluator at that particular evaluator.

The abstract machine of [Clack and Peyton Jones 1987] solves the problems of synchronisation of processes evaluating pieces of the graph on a shared memory architecture when only the evaluator ξ_1 is being used. In [Karia 1987] an abstract distributed memory architecture is defined which fully supports the evaluation transformer model of parallel reduction. This is in the process of being further refined [Bevan et al 1987].

7. Other Issues.

For expository purposes, we have made some restrictions in the main body of the paper. We give an indication of how these may be lifted in this section.

We have restricted ourselves to discussing the evaluators ξ_0 to ξ_3. The evaluators ξ_2 and ξ_3 were chosen because they treated each element of the list in a uniform way. It is shown in [Wadler 1987], [Burn 1987a] that we can choose abstract domains for other evaluators which treat the elements of a list in a uniform way, and also for other recursive data types. For example, if we had a list of lists, then we may want to evaluate the structure of the top level list and to evaluate each of the elements of the list, themselves lists, using ξ_3. We can cope with these by adding extra points in the abstract domain for lists [Wadler 1987], [Burn 1987a]. The evaluation transformer theorems of [Burn 1987a] still hold in these cases. Furthermore, in [Burn 1987a] abstract domains are given for data types constructed by finite applications of sums, products and lifting to base types, and the evaluation transformer theorems hold for these situations as well.

The strictness information and evaluation transformers we have been discussing are really what are called *context-free* strictness [Hankin, Burn and Peyton Jones 1986], where the problem was first isolated, and *context-free* evaluation transformers [Burn 1987a], for they are valid in any context in which the function applied. When we have higher-order functions in our language, then we can sometimes find out more information which will allow stronger evaluators to be used on an argument in an application. For example, it can be determined that

$$g = \lambda f.\lambda x.f\ x$$

is strict in its first argument in any application, but not in its second. (Use Theorem 4-1 to show this.) However, if g is applied to a strict function, then it also needs to evaluate its second argument. Thus taking into account this second type of contextual information means that in this case we are able to evaluate the second argument to g. Such *context-sensitive* issues are covered in [Hankin, Burn and Peyton Jones 1986] in the case of strictness analysis, and in [Burn 1987a] for the more general issue of evaluation transformers.

Finally we note that although we have been discussing parallel evaluation in this paper, evaluation transformers can be used in a sequential system as well, for the semantic criterion is exactly that which will ensure the same semantics as lazy evaluation. Instead of evaluating an argument sometime in the future, we are able to evaluate the argument when the function is applied to it, passing its value rather than creating a closure. It remains a matter for experiment whether it is worth using the evaluators ξ_2 and ξ_3 because they create the whole of the structure of a list which may overload memory. In a parallel system this is hopefully not so much of a problem because it is possible that consumers of a list are using the list in parallel with its creation. Simple strictness information has been used to optimise code for the G-machine [Johnsson 1987], a sequential architecture for graph reduction. In some cases this has been responsible for an order of magnitude increase in the speed of executing programs. This is because expressions can sometimes be evaluated on the stack instead of writing them out into graph store.

8. Relationship to Other Work.

Part of the motivation for this work came from thinking about [Hughes 1985], as well as realising that we were not making full use of the abstract interpretation that we had. We will restrict our discussion in this section to relating our work and that contained in [Wadler and Hughes 1987] (also appearing in this volume), which has developed further the ideas of [Hughes 1985], although we note that [Hall and Wise 1987] bears some similarity to [Wadler and Hughes 1987].

We were interested to read the paper by Wadler and Hughes because it is tackling the same problem and gives a similar final solution, but uses a different route to get there. On the surface, the only differences seem to be that this paper is able to deal with higher-order functions, while [Wadler and Hughes 1987] can find out information about more contexts. However, there are some quite subtle differences. Unfortunately we must

content ourselves with listing some of these differences and having an initial guess at their implications, for it appears that a lot more work must be done in order to relate the two pieces of work more adequately.

In [Wadler and Hughes 1987] the domain theoretic concept of a *projection* [Scott 1976] is brought to bear in order to describe the behaviour of functions. (The terms projection and *context* are used interchangeably in [Wadler and Hughes 1987].) Of particular interest to us are the projections T' and $H'\lceil|T'$ from the paper of Wadler and Hughes. T' projects all lists with an undefined tail to $\perp$ and is the identity everywhere else. Any list which has an undefined tail or an undefined element is mapped to $\perp$ by the projection $H'\lceil|T'$. If f is the semantics of some function, and it satisfies the condition

$$f \circ T' = f$$

then the function is called *tail-strict*. The analagous concept for $H'\lceil|T'$ is head and tail strict. Intuitively it would seem to be the case that if a function is tail strict, then its argument can be evaluated using a tail strict evaluator.

If α and β are contexts, then a function f is β-strict in an α-strict context if

$$\beta \circ f \circ \alpha = f,$$

that is, if it is safe to evaluate an application of f in a β-strict context, then it is safe to evaluate the argument of f in an α-strict context.

Initially we thought that evaluators corresponded directly to *contexts* (or *projections*) and evaluation transformers to *context transformers* (or *projection transformers*); in particular, the evaluators ξ_2 and ξ_3 corresponded respectively to the contexts T' and $H'\lceil|T'$. We were surprised therefore in talking with Phil Wadler when we realised that the evaluators ξ_2 and ξ_3 are not the same as the projections T' and $H'\lceil|T'$. For example, T' projects any list with an undefined tail onto the bottom element, while ξ_2 preserves the denotational semantics of the list. Rather, it seems to be the case that if the analysis of [Wadler and Hughes 1987] allows the evaluation of a list in the context T', then it is safe to evaluate the list using the evaluator ξ_2. A similar relationship holds between ξ_3 and $H'\lceil|T'$.

This brings us to where the fundamental differences in the two approaches seem to lie. Our work relates the denotational semantics and the operational semantics of a program. The semantic criterion which we use for changing the evaluation strategy (see section 1) relates the denotational and operational semantics of a program. We saw in section 3 that the points in the abstract domain were chosen because they modelled the

behaviour of evaluators. Facts 4-1 and 5-1 use the denotational semantics to say which evaluator is safe for an argument to a function and they are proved by discussing the operational behaviour of the evaluators. Contexts however are a denotational notion. The tests given in [Wadler and Hughes 1987] see whether disturbing the denotational semantics of an argument, for example, sending a list which has an undefined tail to bottom using T', has any effect on the denotational semantics of the result of a function application. If, for example, first applying T' to the argument list has no effect on the semantics of the function, then this is interpreted to mean that a tail strict *cons* can be used (i.e. use the evaluator ξ_2), but no formal proof is given which allows movement from changing the denotational behaviour to changing the operational behaviour. In fact, it is in [Burn 1987a], by explicitly relating the denotational and operational behaviour, that the first formal proof is given that any of the extant analyses can be used to allow a change in the operational behaviour of the evaluation of a functional program.

Care must be taken when comparing the two pieces of work because they have two different notions of safety. Safety here relates the denotational and operational behaviour of programs, whereas safety in [Wadler and Hughes 1987] is again a denotational notion.

It is interesting to note that the denotational semantics of the evaluators ξ_2 and ξ_3 are just the identity function, that is, they satisfy the conditions for the identity projection *ID* of [Wadler and Hughes 1987]. This is curious because one may therefore wish to conclude from their analysis that it is therefore safe to use the evaluators ξ_2 and ξ_3 in any function application, which is not true. It is clear that if an expression is to be evaluated in the context *ID*, then this is intuitively supposed to indicate that no evaluation is allowed (i.e. use ξ_0). Therefore it is not safe just to implement any evaluator which has the same denotational semantics as a projection. Anyway, one would not wish to be constrained to choosing an evaluator which modelled the behaviour of a projection. For example, the T' projection would imply that all the list must be evaluated before it is available to any other process, for T' must return $\perp$ if the list has an undefined tail. (Recall that uses of ξ_2 and ξ_3 allow the list to be consumed as each *cons* cell is produced.) There is room here for some more work investigating the relationship between the information gained from the context analysis and the operational behaviour of an evaluator.

There are many other relationships between the two pieces of work. Our analysis is based on abstract interpretation, while [Wadler and Hughes 1987] is based on projections. There are many more projections discussed in their paper than we discuss here, and so one must determine whether one analysis is able to give more accurate information than the other and whether one is able to give a greater variety of information. As we pointed out

in section 7, we have restricted ourselves to the evaluators ξ_0 to ξ_3 for expositional purposes. If one is interested in finding out more information, then the abstract domains can be extended to include points which model the behaviour of the evaluators. However we will not be able to capture all notions of evaluators in this way. It is our opinion that we will not be able to capture the notion of head strictness presented in [Wadler and Hughes 1987] in an abstract interpretation, for it seems to need abstract domains with an infinite amount of information (i.e. infinite number of points), which would make the abstract interpretation uncomputable. So in this sense, the analysis of [Wadler and Hughes 1987] is the more powerful. However, we have isolated another notion of head strictness which one is able to determine using abstract interpretation, and which appears at the moment to fit naturally into our evaluation transformer model of parallel reduction [Burn 1987b].

Our analysis is able to deal with higher-order functions, whereas [Wadler and Hughes 1987] is unable to.

Finally, suppose that all the problems referred to in the above discussion have been solved and that one of the analyses has been used to determine evaluation (or context) transformers. This paper shows how to use the information to give a parallel reduction model of functional languages. One could just compile three different versions of a function, choosing the version according to the context of the application. However we argued in section 5 that it is not possible to determine at compile-time all the contexts in which an expression will occur at run-time. We have therefore proposed a dynamic run-time mechanism. This records the particular context in which an expression is being evaluated, and is then transformed by the evaluation transformers to give the amount of evaluation which is allowed for the arguments in a function application. Pragmatic issues become important at the actual implementation level, for more evaluators mean larger evaluation transformers which mean more complex hardware. Therefore one has to choose a suitably sized set of evaluators which are sensible for parallel evaluation.

9. Conclusion.

With conservative parallel evaluation of functional languages, we must find out which expressions can be evaluated in parallel. A series of papers has developed an abstract interpretation which gives the definedness of functions in terms of the definedness of their arguments. Traditionally this has been used to give a strictness analysis of functions, which has then been interpreted to say when arguments to functions could be evaluated in parallel with applications. However, we have shown in this paper that strictness analysis loses information which is available from the abstract

interpretation because it does not take into account the context of an expression. At compile time we are able to determine from the abstract interpretation what evaluator is safe for the argument in an application given the evaluator which is safe for the application. This leads to a model for the parallel evaluation of functional languages which is a natural generalisation of lazy evaluation. These ideas can also be used to change the evaluation order of sequential implementations.

10. Acknowledgements.

I would like to thank my colleagues at GEC, David Bevan and Rajiv Karia, and my PhD supervisor, Chris Hankin, for many helpful discussions. I also thank Chris Hankin and Simon Peyton Jones for their helpful suggestions about earlier drafts of this paper. Since becoming aware of the work of Phil Wadler and John Hughes I have had many useful discussions with Phil about the relationship between the two pieces of work, which have helped clarify certain ideas and the presentation of this paper, and have highlighted further work to be done.

This work was partially funded by ESPRIT Project 415 : Parallel Architectures and Languages for AIP - A VLSI-Directed Approach.

11. References.

[Abramsky 1985a]

Abramsky, S., Strictness Analysis and Polymorphic Invariance, *Workshop on Programs as Data Objects*, DIKU, Denmark, 17-19 October, 1985, Ganzinger, H., and Jones, N.D., (eds.) Springer-Verlag LNCS 217, pp. 1-23.

[Abramsky 1985b]

Abramsky, S., *Abstract Interpretation, Logical Relations and Kan Extensions*, Draft Manuscript, Imperial College, University of London, October, 1985.

[Bevan et al 1987]

Bevan, D.I., Burn, G.L., Karia, R.J., and Robson, J.D., Design Principles of a Distributed Memory Architecture for Parallel Graph Reduction, *Draft Manuscript*, January, 1987.

[Burn 1987a]

Burn, G.L., *Abstract Interpretation and the Parallel Evaluation of Functional Languages*, PhD Thesis, Department of Computing, Imperial College of Science and Technology, University of London, 1987.

[Burn 1987b]

Burn, G.L., Head-strictness and its Determination Using Abstract Interpretation, *In Preparation*, June 1987.

[Burn, Hankin and Abramsky 1986]

Burn, G.L., Hankin, C.L., and Abramsky, S., Strictness Analysis for Higher-Order Functions, *Science of Computer Programming*, 7, November 1986, pp.249-278.

[Clack and Peyton Jones 1986]

Clack, C., and Peyton Jones, S.L., The Four-Stroke Reduction Engine, *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, Cambridge, Massachusetts, 4-6 August, 1986, pp. 220-232.

[Cousot and Cousot 1979]

Cousot, P., and Cousot, R., Systematic Design of Program Analysis Frameworks, *Conference Record of the 6th ACM Symposium on Principles of Programming Languages*, pp. 269-282, 1979.

[Hall and Wise 1987]

Hall, C.V., and Wise, D.S., Compiling Strictness into Streams, *14th ACM Symposium on Principles of Programming Languages*, pp. 132-143, January, 1987.

[Hankin, Burn and Peyton Jones, 1986]

Hankin, C.L., Burn, G.L., and Peyton Jones, S.L., A Safe Approach to Parallel Combinator Reduction (Extended Abstract), *Proceedings ESOP 86 (European Symposium on Programming)*, Saarbrucken, Federal Republic of Germany, March 1986, Robinet, B., and Wilhelm, R. (eds.), Springer-Verlag LNCS 213, pp. 99-110.

[Hudak and Young 1985]

Hudak, P., and Young, J., Higher-Order Strictness Analysis in Untyped λ-Calculus, *12th ACM Symposium on Principles of Programming Languages*, pp. 97-109, January, 1985.

[Hughes 1985]

Hughes, J., Strictness Detection in Non-Flat Domains *Workshop on Programs as Data Objects*, DIKU, Denmark, 17-19 October, 1985, Ganzinger, H., and Jones, N.D., (eds.) Springer-Verlag LNCS 217, pp. 112-135.

[Johnsson 1987]

Johnsson, T., *Compiling Lazy Functional Languages*, PhD Thesis, Department of Computer Sciences, Chalmers University of Technology, 1987.

[Karia 1987]

Karia, R.J., *An Investigation of Combinator Reduction on Multiprocessor Architectures*, PhD Thesis, University of London, 1987.

[Peyton Jones 1987]

Peyton Jones, S.L., *The Implementation of Functional Programming Languages*, Prentice-Hall International Series in Computer Science, 1987.

[Mycroft 1981]

Mycroft, A., *Abstract Interpretation and Optimising Transformations for Applicative Programs*, PhD. Thesis, University of Edinburgh, 1981.

[Mycroft and Nielson 1983]

Mycroft, A., and Nielson, F., Strong Abstract Interpretation Using Power Domains (Extended Abstract) *Proc. 10th International Colloquium on Automata, Languages and Programming : Springer Verlag LNCS 154*, Diaz, J. (ed.), Barcelona, Spain, 18th-22nd July, 1983, 536-547.

[Scott 1976]

Scott, D., Data Types as Lattices, *SIAM J.Comput. 5* 3 (Sept 1976), 522-587.

[Turner 1985]

Turner, D.A., Miranda: A non-strict functional language with polymorphic types, *Functional Programming Languages and Computer Architecture*, September 1985, Nancy, Jouannaud, J.-P., (ed.), Springer-Verlag LNCS 201, pp. 1-16.

[Wadler 1987]

Wadler, P., *Strictness Analysis on Non-Flat Domains (by Abstract Interpretation over Finite Domains)*, in Abramsky, S., and Hankin, C., (eds), *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987. (Originally distributed on the FP mailboard November, 1987.)

[Wadler and Hughes 1987]

Wadler, P., and Hughes, R.J.M., Projections for Strictness Analysis, *In this volume.*