A Safe Approach to Parallel Combinator Reduction (Extended Abstract).

Conference Paper · March 1986

DOI: 10.1007/3-540-16442-1_7 · Source: DBLP

Chris Hankin
Imperial College London
190 PUBLICATIONS **5,033** CITATIONS

Simon Loftus Peyton Jones
Microsoft
381 PUBLICATIONS **17,780** CITATIONS

A SAFE APPROACH TO PARALLEL COMBINATOR REDUCTION (EXTENDED ABSTRACT)

Chris L. Hankin, Geoffrey L. Burn**, and Simon L. Peyton Jones****

ABSTRACT

In this paper we present the results of two pieces of work which, when combined, allow us to go from a program text in a functional language to a parallel implementation of that program. We present techniques for discovering sources of parallelism in a program at compile time, and then show how this parallelism is naturally mapped into a parallel combinator set that we will define.

To discover sources of parallelism in a program, we use abstract interpretation. Abstract interpretation is a compile-time technique which is used to gain information about a program that may then be used to optimise the execution of the program. A particular use of abstract interpretation is in strictness analysis of functional programs. In a language that has lazy semantics, the main potential for parallelism arises in the evaluation of operands of strict operators. A function is strict in an argument if its value is undefined whenever the argument is undefined. If we can use strictness analysis to detect which arguments a function is strict in, we then know that these arguments can be safely evaluated in parallel because this will not affect the lazy semantics.

Having identified the sources of parallelism at compile-time it is necessary to communicate these to the run-time system. In the second part of the paper we use an extended set of combinators, including a pair of parallel combinators that achieve this purpose.

Keywords: Combinators, Strictness Analysis, Functional Languages, Parallel Reduction, Director Strings

* Department of Computing, Imperial College of Science and Technology, 180 Queen's Gate, London, SW7 2BZ, United Kingdom. Electronic Mail : clh@icdoc.

** GEC Research Ltd, Hirst Research Centre, East Lane, Wembley, Middx, HA9 7PP, United Kingdom. Electronic Mail: geoff@hrc63.

*** Department of Computer Science, University College London, Gower St, London, WC1E 6BT, United Kingdom. Electronic Mail : simonpj@ucl.cs

The first two authors were partially funded by ESPRIT Project 415 - Parallel Architectures and Languages for AIP : A VLSI Directed Approach

1. Mathematical and Notational Preliminaries

A semantically sound method of strictness analysis is given in [Burn, Hankin and Abramsky 1985]. We summarise the approach in this section, using a more perspicuous notation following [Abramsky 1985].

As our language we will use the typed λ-calculus with a set of base types and a set of typed constants. Given base types A, B, $\cdots$, we have type expressions σ, τ defined by

$$\sigma ::= A \mid \sigma \rightarrow \sigma$$

The set of typed constants is denoted by $\{c_\sigma\}$ (e.g. some typed constants are 4_{int}, $+_{int \rightarrow int \rightarrow int}$ and $cons(\perp_A, nil_{Alist})_{Alist}$). For each type σ we have an infinite set of typed variables $Var_\sigma = \{x^\sigma \cdots\}$. Our language of expressions, **Exp**, consists of typed terms $e{:}\sigma$ formed according to the following rules :

(1) $x^\sigma : \sigma$	variables
(2) $c_\sigma : \sigma$	constants
(3) $\dfrac{e : \tau}{\lambda x^\sigma . e : \sigma \rightarrow \tau}$	abstraction
(4) $\dfrac{e_1 : \sigma \rightarrow \tau \quad e_2 : \sigma}{e_1 e_2 : \tau}$	application
(5) $\dfrac{e : \sigma \rightarrow \sigma}{fix\ e : \sigma}$	fixed points

An interpretation, I, is given by

$$I = (\{D_A^I\}, \{c_\sigma^I\})$$

where for each base type A we have that D_A^I is a bounded-complete ω-algebraic cpo [Scott 1981]. This is extended to type $\sigma \rightarrow \tau$ by defining $D_{\sigma \rightarrow \tau}^I$ to be the continuous maps $D_\sigma^I \rightarrow D_\tau^I$. Each c_σ is given interpretation c_σ^I in D_σ^I.

This interpretation induces a semantic function

$$E^I : \mathbf{Exp} \rightarrow Env^I \rightarrow \bigcup D_\sigma^I$$

where $Env^I = \{Env_\sigma^I\}$ and $Env_\sigma^I = Var_\sigma \rightarrow D_\sigma^I$.

$$E^I\,[[x^\sigma]]\,\rho = \rho(x^\sigma)$$

$$E^I\,[[c_\sigma]]\,\rho = c_\sigma^I$$

$$E^I\ [[\lambda x^\sigma . e]]\ \rho \ =\ \lambda y^{D^I_\sigma}\ E^I\ [[e]]\ \rho[y/x^\sigma]$$

$$E^I\ [[e_1\ e_2\ ]]\ \rho \ =\ (E^I\ [[e_1]]\ \rho)\ (E^I\ [[e_2]]\ \rho)$$

$$E^I\ [[\mathit{fix}\ e]]\ \rho \ =\ \mathit{fix}(E^I\ [[e]]\ \rho)$$

Throughout the rest of the paper we will have a *standard* interpretation $(\{D_A^{st}\},\{c_\sigma^{st}\})$ where we have the usual flat domains for integers and booleans etcetera and the standard domain for lists. As constants we will have the usual arithmetic and boolean constants, operators such as + and **and**, and a conditional, $\text{if}_\sigma : \mathit{bool} \rightarrow \sigma \rightarrow \sigma \rightarrow \sigma$, for each type σ, which have their usual interpretations. We will call the induced semantic function E^{st}, and we will always use the environment ρ for the standard interpretation.

For the abstract interpretation, we will use the domain $2 = \{0,1\}$, with $0 \leqslant 1$, for the interpretation of all base types A (including lists). We call the induced semantic function E^{ab} and use ρ' for the environment in the abstract interpretation.†

We relate the standard and abstract interpretations, by defining an abstraction map so that we can show that calculations in the abstract domain correctly model calculations in the standard domain.

$$\begin{tikzcd} & \mathbf{Exp} \arrow[dl, "\textit{standard interpretation}"'] \arrow[dr, "\textit{abstract interpretation}"] & \\ D^{st} \arrow[rr, "abs"'] & & D^{ab} \end{tikzcd}$$

For strictness analysis, this is done by defining on the base type A the map

$$abs_A\ : D_A^{st} \rightarrow D_A^{ab}$$

$$abs_A(a) \ =\ \begin{cases} 0 \text{ if } a = \bot_{D_A^{st}} \\ 1 \text{ otherwise} \end{cases}$$

We can then define abstraction maps for each of the finite higher types in a natural way. Details can be found in [Burn, Hankin and Abramsky, 1985].

† Note that in [Burn, Hankin and Abramsky 1985] the standard interpretation of the types was called D_σ and the abstract interpretation of the types was called B_σ. Also, the standard semantic function was called *sem*, while the abstract semantic function was called *tabs*.

Having defined the abstraction maps for each finite type, we are then able to set the abstract interpretation of the constants to be the abstraction of the standard interpretation.

The main result of [Burn, Hankin and Abramsky, 1985] was the following *Soundness Theorem for Strictness Analysis*, which says that if something is strict in the abstract interpretation then it is strict in the standard interpretation.

Theorem 1.1:

Given $f : \sigma \to \tau$, interpretations of constants such that $abs_{\sigma}(E^{st}[[c_{\sigma}]]\rho) \leqslant E^{ab}[[c_{\sigma}]]\rho'$ for all constants c_{σ}, and environments ρ and ρ' satisfying $abs_{\tau}(\rho(x^{\tau})) \leqslant \rho'(x^{\tau})$ for all x^{τ}, we have that

$$(E^{ab}[[f]]\rho')\perp_{D_{\sigma}^{ab}} = \perp_{D_{\tau}^{ab}} \Rightarrow (E^{st}[[f]]\rho)\perp_{D_{\sigma}^{st}} = \perp_{D_{\tau}^{st}}. \qquad \square$$

The strictness analysis of [Burn, Hankin and Abramsky 1985] allows the detection of when arguments are definitely needed by functions, that is, the possibility of strict applications. When we write out an expression in the text, we will indicate the strict applications by writing # in between the operator and the operand.

2. Pragmatics of Strictness Analysis

The original work on abstract interpretation of applicative programs [Mycroft 1981] treated first order recursion equations over flat domains. In this situation it was sufficient to find out the strictness of a function in each of its arguments, for this strictness information was constant in all contexts in which the function was used. This strictness information is referred to as *context-free strictness* in this paper.

Definition 2.1:

A function $f : \sigma_1 \to \cdots \to \sigma_{n+1}$ is *context-freely strict* in its *ith* parameter if for all $s_j \in D_{\sigma_j}^{st}$, $j \neq i$, we have that

$$(E^{st}[[f]]\rho)\, s_1 \cdots s_{i-1} \perp_{D_{\sigma_i}^{st}} s_{i+1} \cdots s_n = \perp_{D_{\sigma_{n+1}}^{st}} \qquad \square$$

When we introduce higher-order functions, the strictness information for particular arguments to a function may change according to the context in which the function appears. For instance, taking the apply function :

$$g = \lambda f^{A \to A}.\lambda x^{A}.f(x)$$

we can see that if g is used in the context $g \# f\, e$ and f is a strict function, then the apply node between $g \# f$ and e could be changed to # because $g \# f$ needs the value of e.

This issue was addressed informally in [Burn, Hankin and Abramsky 1985]. In the current paper we refer to this strictness information as *context-sensitive strictness.*

Definition 2.2:

A function $f : \sigma_1 \rightarrow \cdots \rightarrow \sigma_{n+1}$ is *context-sensitively strict* in its *ith* parameter in an application $f\, e_1 \cdots e_n$ if

$$(E^{st}\,[[f]]\,\rho)\, E^{st}\,[[e_1]]\,\rho \cdots E^{st}\,[[e_{i-1}]]\,\rho \perp_{D^{st}_{\sigma_i}} E^{st}\,[[e_{i+1}]]\,\rho \cdots E^{st}\,[[e_n]]\,\rho = \perp_{D^{st}_{\sigma_{n+1}}}$$

□

The pragmatics presented in [Burn, Hankin and Abramsky 1985] have two shortcomings which are remedied below.

2.1. Context-Free Strictness Information

The first shortcoming of the abovementioned paper is the lack of consideration of context-free strictness, with the resultant loss of information, and which can be seen by looking at an example involving a higher-order conditional :

(if condition then f_1 else f_2) expensive

where f_1 is strict and f_2 is non-strict. If the *condition* is *true*, then we would like to be able to reduce the above expression to f_1 *# expensive*, which would allow the opportunity for parallel evaluation, while if it is *false*, we have to reduce the conditional to f_2 *expensive*.

This problem can be solved if we associate the context-free strictness information with a function so that it is available in any application. For functions involving functional arguments, this context free strictness information may be weaker than the strictness information available in some particular contexts.

Theorem 2.3 introduces a test for context-free strictness.

Theorem 2.3:

If $f : \sigma_1 \rightarrow \cdots \sigma_n \rightarrow A$ and

$$(E^{ab}\,[[f]]\,\rho')\, \top_{D^{ab}_{\sigma_1}} \cdots \top_{D^{ab}_{\sigma_{i-1}}} \perp_{D^{ab}_{\sigma_i}} \top_{D^{ab}_{\sigma_{i+1}}} \cdots \top_{D^{ab}_{\sigma_n}} = 0$$

then f is context-freely strict in its *ith* parameter.

□

2.2. Context-Sensitive Strictness Information

In [Burn, Hankin and Abramsky 1985] only information from the first $i-1$ arguments in an application was used when determining the context-sensitive strictness of a function in its *ith* argument. This method has its limitations which can be illustrated by considering the *apply* function, g, defined earlier. If we apply g to $\lambda x^A.x$, which is strict, then we are able to label the apply nodes in the application $g \ \# \ (\lambda x^A.x) \ \# \ \mathit{expensive}$ as before. However, if we were to reorder the parameters to g, defining g' :

$$g' = \lambda x^A.\lambda f^{A \to A} \ f(x)$$

then we can no longer label the apply node involving the expensive computation in the call $g' \ \mathit{expensive} \ \# \ (\lambda x^A.x)$. Thus the labelling of the apply nodes is different because of the different ordering of the parameters. This is clearly unsatisfactory. It would be much better if we could label apply nodes in an application using the information about the other arguments in the application; this is formalised in Theorem 2.4.

Theorem 2.4:

Given $f : \sigma_1 \to \cdots \to \sigma_{n+1}$ and an application $f \, e_1 \cdots e_n$, if

$$E^{ab}[[f]]\rho' \, E^{ab}[[e_1]]\rho' \cdots E^{ab}[[e_{i-1}]]\rho' \perp_{D^{ab}_{\sigma_i}} E^{ab}[[e_{i+1}]]\rho' \cdots E^{ab}[[e_n]]\rho' = \perp_{D^{ab}_{\sigma_{n+1}}}$$

then f is context-sensitively strict in its *ith* argument in the application $f \, e_1 \cdots e_n$.

□

2.3. Lifting Out Free Variables and the Conditional

A third way in which we can improve the amount of available parallelism is again in the conditional. Consider a conditional $\lambda y^A.\mathit{if\ condition\ then}\ e_1\ \mathit{else}\ e_2$ where neither the *condition* nor e_2 needs the value y, but e_1 does. Then if the value of the *condition* is *true*, we can trigger the evaluation of y. This effect can be obtained by λ-abstracting the variable y from the expression e_1 to obtain $(\lambda y.e_1)\,y$ for the expression e_1. This enables us to label the applications as follows :

$$\lambda y^A.\mathrm{if} \ \# \ \mathit{condition} \ ((\lambda y^A.e_1) \ \# \ y) \ e_2$$

More generally, if $\{x_1, \cdots, x_n\}$ are the free variables of a branch of the conditional, e_i, then we can convert it to $f \, x_1 \cdots x_n$ where $f = \lambda x_1 \cdots \lambda x_n.e_i$.

In a similar manner, if we are given an n-ary constant function which is not strict in certain arguments, a **parallel or** for example, then we can apply the above transformation to get maximum parallelism.

3. Parallel Combinators

In this section we define a set of "parallel" combinators. It is intended that these should be used as the basis for the machine code of a machine such as the COBWEB machine [Hankin, Osmon and Shute, 1985]. An essential requirement is that the combinator code should be capable of encoding the two types of strictness information described in the last section. Our treatment is based on the director string approach - there is a good match between the director philosophy of viewing combinators as annotations on applications and our requirement to encode strictness annotations on applications.

In [Kennaway and Sleep, 1981] four directors are introduced:

^	send the argument to both operator and operand
/	send the argument to the operator
\	send the argument to the operand
-	destroy the argument

Inner bound variables are abstracted first and occurrences of the bound variable are replaced by **I**; if the variable is not used in a subexpression then it is unaffected by the abstraction algorithm. For example,

$$\lambda x.\lambda y. + \; x \; (\times \; 2 \; y)$$

becomes

$$/\backslash(\backslash + \mathbf{I})\;(\backslash(\times 2)\;\mathbf{I}).$$

Kennaway and Sleep go some way towards pointing out the correspondence between the director strings and Turner's long reach combinators [Turner 1979]. A complete combinator definition of directors can only be given after the introduction of two new combinators [Joy et al. 1985]. The first is a long reach **K** which we call **J** and which has the following reduction rule:

$$\mathbf{J}\;x\;y\;z \;\;=>\;\; x\;y$$

We will also require a long reach version of this which is denoted **J**' in the standard way and which has the following reduction rule:

$$\mathbf{J}'\;a\;x\;y\;z \;\;=>\;\; a\;x\;y$$

The correspondence between directors and combinators is then given by the following table:

^ **S** or **S'**	\B or **B'**
/ **C** or **C'**	− **J** or **J'**

The standard combinators are used to encode the rightmost director on each application and all other directors are represented by long reach combinators. Returning to our example, the combinator equivalent is then

$$\mathbf{C'}\,\mathbf{B}\,(\mathbf{B} + \mathbf{I})\,(\mathbf{B}\,(\times\ 2)\,\mathbf{I})$$

The abstraction process to produce parallel code using this approach can be formalised in the following way. We start by introducing two new combinators **P** and **P**':

$$\mathbf{P}\ f\ x = f\ \#\ x$$
$$\mathbf{P}'\ k\ f\ x = k\ f\ \#\ x$$

P and **P**' are semantically like **I**, but each launches a parallel process (and so we have a natural generalisation of the sequential case, where a sequential evaluator would just treat a **P** as an **I**).

Our strategy is to compile λ-expressions annotated by the strictness analyser into the extended combinator set. We find it convenient to break the compiler into two passes. The first pass, *C*, compiles λ-expressions into an intermediate form, **Dexp**, and the second pass, *G*, compiles **Dexps** into combinator code.

Dexp is a convenient notation which makes explicit the fact that combinators are just annotations on apply nodes. The abstract syntax of **Dexp** is given below, where from now on we will ignore types in order to simplify the presentation.

```
DExp ::= Const
       | Var
       | DExp '<' DirectorString '>' DExp

DirectorString ::=     /* Empty */
                 | Director DirectorString

Director ::= S | B | C | J | P
```

The code generation function, *G*, just converts the code with annotated apply nodes to the standard combinator code.

We will use !, instead of λ, to indicate the parameters in which a function is context-freely strict and an infix # to indicate strict application. Otherwise, our source language is a type-free version of **Exp**, and we will refer to this extended language as **Exp** in the following.

Now we define C:**Exp** $\rightarrow$ **DExp**, which compiles **Exps** to **DExps**. It uses an auxiliary function A:**Var** $\rightarrow$ **DExp** $\rightarrow$ **DExp**, which abstracts a variable from a **DExp**.

We let E, E_1, E_2 denote arbitrary expressions and D denote a possibly empty director string.

$$\begin{aligned}
C[[\lambda x.E]] &= A\ x\ [[C[[E]]]] \\
C[[!x.E]] &= \mathbf{P} <> (A\ x\ [[C[[E]]]]) \\
C[[E_1\ E_2]] &= C[[E_1]] <> C[[E_2]] \\
C[[E_1\ \#\ E_2]] &= C[[E_1]] <\mathbf{P}> C[[E_2]] \\
C[[x]] &= x \\
C[[c]] &= c \\
C[[\mathit{fix}\ e]] &= \mathbf{Y} <> C[[e]]
\end{aligned}$$

$$\begin{aligned}
&A\ x\ [[x]] = \mathbf{I} \\
&A\ x\ [[y]] = \mathbf{K} <> y \\
&A\ x\ [[E_1 <D> E_2]] \\
&\quad = (A\ x\ [[E_1]]) <\mathbf{S}D> (A\ x\ [[E_2]]) && x \in E_1\ \&\ x \in E_2 \\
&\quad = (A\ x\ [[E_1]]) <\mathbf{C}D> E_2 && x \in E_1\ \&\ x \notin E_2\ \&\ D \neq \mathbf{P}D' \\
&\quad = (A\ x\ [[E_1]]) <\mathbf{P}\mathbf{C}D'> E_2 && x \in E_1\ \&\ x \notin E_2\ \&\ D = \mathbf{P}D' \quad (*) \\
&\quad = E_1 <\mathbf{B}D> (A\ x\ [[E_2]]) && x \notin E_1\ \&\ x \in E_2 \\
&\quad = E_1 <\mathbf{J}D> E_2 && x \notin E_1\ \&\ x \notin E_2\ \&\ D \neq \mathbf{P}D' \\
&\quad = E_1 <\mathbf{P}\mathbf{J}D'> E_2 && x \notin E_1\ \&\ x \notin E_2\ \&\ D = \mathbf{P}D' \quad (*)
\end{aligned}$$

The two rules (*) transform

$E_1 <\mathbf{C}\mathbf{P}D> E_2$ to $E_1 <\mathbf{P}\mathbf{C}D> E_2$

(and the same for **J**) which is correct since:

$$\mathbf{C}'\ (\mathbf{P}'\ D)\ f\ g\ x = \mathbf{P}'\ D\ (f\ x)\ g = D\ (f\ x)\ \#\ g$$

$$\mathbf{P}'\ (\mathbf{C}'\ D)\ f\ g\ x = \mathbf{C}'\ D\ f\ \#\ g\ x = D\ (f\ x)\ \#\ g$$

but the second allows us to initiate the parallel process to evaluate g at an earlier stage.

We now present a code generation scheme G, which translates **DExps** to combinator expressions, where d denotes any director :

$$\begin{aligned}
&G[[E_1 <> E_2]] = G[[E_1]]\ G[[E_2]] \\
&G[[E_1 <D> E_2]] = H[[D]]G[[E_1]]\ G[[E_2]] \\
&G[[x]] = x
\end{aligned}$$

$$\begin{aligned}
&H[[d]] = d \\
&H[[dD]] = d'(\ H[[D]])
\end{aligned}$$

H is an auxiliary function which compiles the director strings.

Returning to our earlier example, we find that the function is context-freely strict in both arguments and all applications are strict and thus we generate the following code :

$$\mathbf{P}\,(\mathbf{B}\;\mathbf{P}\,(\mathbf{C}'(\mathbf{B}'\;\mathbf{P}\,)\;(\mathbf{B}'\;\mathbf{P}\;+\;\mathbf{I})\;((\mathbf{B}'\;\mathbf{P}\,)\;(\mathbf{P}\;\times\;2)\;\mathbf{I})))$$

It is possible to optimise this code, reducing its size and the number of reductions necessary to evaluate it, by combining the **P** combinator with the adjacent director :

$$\begin{array}{ll}
\mathbf{SP} = \mathbf{S}'\,\mathbf{P} & \mathbf{SP}'D = \mathbf{S}'(\mathbf{P}'D) \\
\mathbf{BP} = \mathbf{B}'\,\mathbf{P} & \mathbf{BP}'D = \mathbf{B}'(\mathbf{P}'D) \\
\mathbf{CP} = \mathbf{C}'\,\mathbf{P} & \mathbf{CP}'D = \mathbf{C}'(\mathbf{P}'D) \\
\mathbf{JP} = \mathbf{J}'\,\mathbf{P} & \mathbf{JP}'D = \mathbf{J}'(\mathbf{P}'D)
\end{array}$$

thus introducing eight new combinators (those appearing on the left-hand side of the equals sign), where an example of a reduction rule is :

$$\mathbf{BP}\;f\;g\;x\;=\;f\;\#\;(g\;x)\;=\;\mathbf{B}'\,\mathbf{P}\;f\;g\;x$$

The code generated for the example then becomes :

$$\mathbf{P}\,(\mathbf{B}\;\mathbf{P}\,(\mathbf{C}'\;\mathbf{BP}\,(\mathbf{BP}\;+\;\mathbf{I})\;(\mathbf{BP}\,(\mathbf{P}\;\times\;2)\;\mathbf{I})))$$

4. Relation With Other Work

There are two projects that are closely related to the work reported in this paper. The more interesting is the work of Oberhauser and Wilhelm [Oberhauser and Wilhelm 1984]. Work has also been done by Meira at the University of Kent [Meira 1985].

Oberhauser and Wilhelm present a combinator abstraction algorithm that produces code that closely resembles the director-based combinator code. However an important difference is that they perform flow analysis on the combinator graph to determine strictness rather that on the source program. Since combinators appear as leaves in their approach, rather than as annotations on apply nodes, they produce extra annotations which we claim are redundant. Oberhauser and Wilhelm do have two different types of strictness annotation but these are not exactly the context free and context sensitive annotation presented here and it is not clear that their approach is fully higher-order. Finally their work is based on different semantic foundations from that described in this paper [Maurer 1985].

Meira also directly analyses the combinator code. However he uses a more traditional abstraction algorithm and this means that there is less similarity between his code and the director-based code. He also fails to make the distinction between context-free and context-sensitive strictness so again it seems unlikely that he would identify as much potential for parallelism as we do.

5. Conclusion

In this paper we have presented a method for translating a program in a functional language to a parallel implementation of that program, while retaining lazy semantics. This has involved extending the pragmatics of strictness analysis which is used to discover sources of parallel evaluation. We have also presented a natural way of defining a parallel combinator set, based on the notion that combinators are directors.

While the work gives maximum information about higher-order functions over flat base domains, it gives poor information about functions over recursively defined data types, for at the moment it only supports mapping them down to the two point domain. This clearly needs some more work, and [Hughes 1985] and [Wadler 1985] are first steps in this direction.

6. Acknowledgements

Some of the work presented in this paper is the result of our continued and enjoyable collaboration with Samson Abramsky, to whom we gratefully extend our thanks. David Bevan and Rajeev Karia are also due thanks for their interest and useful comments on the various drafts of the paper.

The work of the first two authors was partially funded by ESPRIT Project 415 - Parallel Architectures and Languages for AIP : A VLSI Directed Approach.

7. References

Abramsky, S., *Abstract Interpretation, Logical Relations and Kan Extensions*, Draft Manuscript, 1985.

Burn, G.L., Hankin, C.L., and Abramsky, S., Strictness Analysis for Higher-Order Functions, To appear in *Science of Computer Programming* Also : *Imperial College of Science and Technology, Department of Computing, Research Report DoC 85/6, April 1985.*

Hankin, C.L., Osmon, P.E., and Shute, M.J., COBWEB: A Combinator Reduction Architecture, in : *Proceedings of IFIP International Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, 16-19 September, 1985, Jouannaud, J.-P. (ed.), Springer-Verlag LNCS 201, pp. 99-112.

Hughes, J., *The Design and Implementation of Programming Languages*, PhD Thesis, Oxford University, 1983. (Published as Oxford University Computing Laboratory, Programming Research Group, Technical Monograph PRG-40, September, 1984.)

Hughes, J., Strictness Detection in Non-Flat Domains *Workshop on Programs as Data Objects*, Copenhagen, Denmark, 17-19 October, 1985. (Proceedings to be published in *Springer Verlag LNCS* series.)

Joy, M.S., Rayward-Smith, V.J. and Burton, F.W., Efficient Combinator Code, *Computer Languages 10* 3, 1985, pp.211-223.

Kennaway, J.R., and Sleep, M.R., *Director Strings as Combinators*, University of East Anglia Report, 1981.

Maurer, D., Strictness Computation Using Generalised λ-expressions, *Workshop on Programs as Data Objects*, Copenhagen, Denmark, 17-19 October, 1985. (Proceedings to be published in *Springer Verlag LNCS* series.)

Meira, S.R. de L., *On the Efficiency of Applicative Algorithms*, PhD Thesis, The University of Kent at Canterbury, March 1985.

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

Oberhauser, H.-G., Wilhelm, R., *Flow Analysis in Combinator Implementation of Functional Programming Languages*, Technical Report, Universitat des Saarlandes, D-6600 Saarbrucken, 1984.

Scott, D., *Lectures on a Mathematical Theory of Computation*, Tech. Monograph PRG-19, Oxford Univ. Computing Lab., Programming Research Group, 1981.

Turner, D.A., Another Algorithm For Bracket Abstraction, *The Journal of Symbolic Logic 44* 2, June 1979, pp. 267-270.

Wadler, P., *Strictness Analysis on Non-Flat Domains (by Abstract Interpretation over Finite Domains)*, Draft Manuscript, 1985.