An Implementation of Static Functional Process Networks

Stuart Cox
Shell-Ying Huang*
Paul Kelly
Junxian Liu
Frank Taylor

Department of Computing, Imperial College, London SW7 2BZ, UK
Tel: +44 71 589 5111 x5028, Fax: +44 71 581 8024
email: phjk@doc.ic.ac.uk

Keywords: functional programming, process network, declarative annotation, transputer, program transformation

Abstract

To get high performance on a distributed-memory multicomputer at present and for the foreseeable future, some explicit control is needed. This paper describes work aimed at harnessing the power of the functional notation in exercising such control. We have developed a declarative annotation scheme which allows explicit control over process placement and communications. The language, called Caliban, has been implemented on a configurable, loosely-coupled commercial multicomputer and we describe the compiler and run-time system.

Introduction

Functional languages have long been the subject of research into automatic parallelisation, because of transparent data dependency, and the freedom with which evaluation order can be changed while retaining determinacy. This work has achieved some success, exemplified most notably by Goldberg's Buckwheat implementation [Gol88]. Attempts to do the same using distributed memory hardware have proven

*School of Applied Science, Nan Yang Technological University, Nan Yang Avenue, Singapore 2263.

less successful, being very sensitive to locality properties which are very hard to evaluate at compile time. This experience is shared by researchers parallelising imperative languages (see for example [ZC90]).

Meanwhile, the functional approach has very powerful abstraction mechanisms, in particular (see Hughes [Hug84]) streams and higher-order functions. These provide substantial leverage on software engineering problems in general. Our objective is to harness this power to the special software engineering problems of distributed memory parallel programming.

In an earlier work [Kel89, chapter 5], a system of annotations was introduced to enable the programmer to control some aspects of a program's execution on a distributed memory machine. The language retains some abstraction: prescription is in terms of a "process network" showing processes evaluating named expressions, linked by arcs showing where communication occurs. The network is described by a declarative annotation of its structure. Furthermore, the host functional language can itself be used to generate the annotation. Functions which do this can be thought of as "network forming operators", which capture a communications *pattern*, and are parameterised by the size and shape needed for a particular application.

This paper describes our attempt at implementing the language. We confine our attention to programs whose process network is independent of run-time parameters (but can be configured at compile-time). This enables us to perform optimised placement of processes in the distributed-memory machine, and eliminates dynamic creation of communications channels.

Section 1 introduces the annotation language and gives a small example of its use. Section 2 discusses our implementation and it's limitations. Sections 3 to 6 describe each of the phases of the compiler and run-time system.

1 Caliban

Caliban is an annotation language which can be built on top of many different languages. The implementation described here is based on a version of Haskell [HWe90] which is being developed at Imperial College in collaboration with Southampton University. We are implementing a simplified version of the annotation language presented in [Kel89, Chapter 5].

1.1 Introduction

To start with, we present a quick tour of the Caliban annotation language.

- In a process network, processes (nodes) are connected by streams of data (arcs).
- Streams are head-strict lazy lists. This ensures that only normal form values are ever sent between nodes. We use the standard list constructors because in our experience providing a special set of stream constructors is obtrusive.

- Streams are all of type [Message]. Message is defined as:

  ```
  data Message = INT Int | CHAR Char | FLOAT Float | BOOL Bool
  ```

- Using a **moreover** clause we can partition the program into separate node expressions, each to be evaluated in parallel on separate PEs. e.g.

  $e_1 = f \ldots e_2 \ldots$

  $e_2 = f \ldots e_3 \ldots$

  moreover

  Node e_1 And Node e_2

- Using data dependency analysis we can determine the need for communication between two nodes. In the previous code we see that e_1 uses e_2 and therefore a connection would have to be set up between e_1 and e_2.

- We can add some structure to the annotation by explicitly documenting which nodes communicate (in either direction) with which others. This is done by adding Arc annotations. e.g. Node e_1 And Node e_2 And Arc e_1 e_2

  These are only used for consistency checking. The compiler can derive all the information it needs from just the Node annotations.

- Each process can now be evaluated eagerly by it's PE. This can continue until an inter-processor data dependency causes blocking or the output buffer space of the PE is filled.

1.2 Example: parallel search

We illustrate this approach with a brief example. Suppose we have a database of records, and a stream of incoming queries. We need to look each query up in the database, and produce as output the stream of corresponding records.

As a specification, here is the sequential formulation:

```
search keys db = map (match db) keys
```

We assume the existence of a database inquiry function match which finds the record corresponding to a single key, or returns FAILURE.

As a simple example of a parallel search implementation, we will subdivide the database into several equal-sized parts, and search for each key in the sub-databases using the original sequential algorithm. Figure 1 shows the process network, with each node labelled with the expression it returns.

This computation is specified by the following definition, where search′ is the parallel version, and search is the sequential version used to search the sub-databases:

```
search' keys db = merge splitresults
                  where
                  splitresults = map (search keys) (split N db)
```

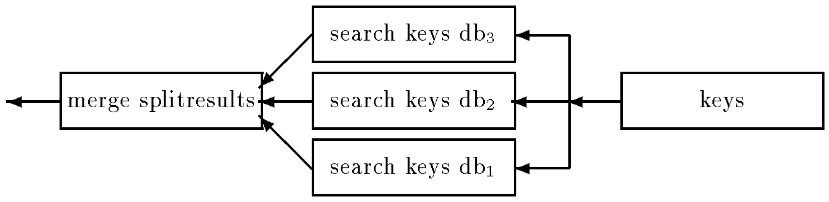

Figure 1: Process network for parallel search implementation

where split N db divides the database into N equal-sized partitions, and merge selects a successful match, if possible, from each set of N replies.

1.2.1 Expressing the parallelism

The next step is to add a **moreover** clause, to specify how the computation is to be distributed. This is straightforward if we know N so that the elements of the list splitresults can be given names. For example, if N = 3, as illustrated in Figure 1, then we can write:

```
search' keys db
      = replies
        where
        replies = merge [splitresult₁, splitresult₂, splitresult₃]
        [splitresult₁, splitresult₂, splitresult₃] = splitresults
        splitresults = map (search keys) (split 3 db)
        moreover
        Node splitresult₁ And Arc replies splitresult₁ And
        Node splitresult₂ And Arc replies splitresult₂ And
        Node splitresult₃ And Arc replies splitresult₃ And Node replies And
        Node splitresult₁ And Arc keys splitresult₁ And
        Node splitresult₂ And Arc keys splitresult₂ And
        Node splitresult₃ And Arc keys splitresult₃ And Node keys
```

This is somewhat long-winded, and changing N is very hard work. We can make life very much easier by defining a function to construct the required annotation. This ordinary function definition defines a "network forming operator" ("NFO"), which abstracts the pattern we want:

```
fan a [ ] = Node a
fan a (b : bs) = Node b And Arc a b And fan a bs
```

If we give this function our list of processes, it will construct the combination of Arc and Node assertions we need. e.g.

```
fan replies [splitresult₁, splitresult₂, splitresult₃]
```

To write the parameterised version of search′ we need to use fan twice — once for the fan-out to the sub-databases, and then again to route the replies to the merge computation:

```
search' keys db = replies
                  where
                  replies = merge splitresults
                  splitresults = map (search keys) (split N db)
                  moreover
                  fan replies splitresults And fan keys splitresults
```

This example illustrates how a function like fan captures a family of parallel computation structures, leading to a concise explanation of how the computation is mapped onto the underlying hardware.

It also demonstrates that some care is needed with the naming of expressions: the name replies labels the process which performs the merge computation. Although slightly counter-intuitive, it is necessary to identify a process by the *value* it computes rather than the *function* it performs to avoid ambiguity.

2 Implementation

In the general case, a **moreover** clause can decorate any expression in the program, so the process network's development can be inextricably intertwined with execution of the program. The aim of the Caliban approach is to allow the programmer to collect the annotations controlling a parallel computation into a single declarative description, to aid the human reader and to provide the implementation with the opportunity to make scheduling decisions based on the needs of the computation as a whole rather than having to allocate each process as it is created.

In the static implementation described here, we examine the extreme case where the entire process network can be elaborated at compile time. This illuminates some of the problems a more ambitious implementation would have to deal with. It also makes it easy to use the best possible embedding of the required process network in the available processor network.

2.1 Limitations of the implementation

The implementation described here imposes quite severe restrictions on the form of programs that can be accepted:

- There can be only one **moreover** clause, although it need not necessarily appear at the outermost level of the program.

- The process network described by the **moreover** must be compile-time static.
- Each PE has just one outgoing stream, which may be sent to several other PEs.

  This is a consequence of the type restriction on the Node operator above. It has the very desirable effect that each PE has only one reduction process: evaluation of two or more output streams would have to be interleaved fairly to respect the desired semantics.

In the following sections we describe each phase of the full compiler.

3 Simplifier

In essence the simplifier's responsibility is to evaluate the annotation, and reduce it to an annotation of the form

Node e_1 And Node e_2 And ... And Node e_k And Arc e_i e_j And ...

where the e_i are the processes to be placed on each PE. We will refer to this as *annotation normal form.*

In particular, an annotation in which a network forming operator f is applied to an expression e must be reduced until it has the above form. This will normally involve rewriting the argument e — but other references to e will probably appear in the body of the computation. When the simplifier evaluates e to expose a constructor the other references to e must be updated to refer to the updated value.

3.1 Example

Consider the search example given earlier:

```
search' keys db = replies
                  where
                  replies = merge splitresults
                  splitresults = map (search keys) (split N db)
                  moreover
                  fan replies splitresults And fan keys splitresults
```

Suppose we know that N is 2 and that split 2 db yields the two-element list [db_1, db_2]. The annotation evaluates to:

Node (search keys db_1) And Arc replies (search keys db_1) And
Node (search keys db_2) And Arc replies (search keys db_2) And Node replies And
Node (search keys db_1) And Arc keys (search keys db_1) And
Node (search keys db_2) And Arc keys (search keys db_2) And Node keys

During the partial evaluation we reduced splitresults to a two-element list, by unfolding the application of map. The problem is that the computational part of the program still refers to splitresults.

3.2 Term-graph rewriting

This difficulty is frustrating since the objective seems straightforward: we wish the annotation to refer to the expressions which *will* appear in the computation. Fortunately, the intuitive understanding of how expressions are generated and shared in a functional language implementation has a tidy formal basis as graph rewriting. Barendregt et al. [BvEG$^+$87] show that term-graph rewriting is a correct implementation of term rewriting as required by our host functional language. Thus we can use their theory to perform straightforward simplification in the annotation, whilst keeping precise track of how expressions are shared with the computation part of the program. We inherit their correspondence theorem with the term-rewrite system, and therefore can be assured that the simplification process does not compromise the correctness of the program's result.

3.3 Termination

The simplification phase may fail to terminate even when the unannotated program would terminate properly. The termination condition is, however, relatively simple as it is independent of the sharing detail: the simplification phase terminates provided ordinary evaluation of the annotation to annotation normal form terminates.

Compile-time simplification may, of course, fail if parameters on which the annotation's normal form depends are unavailable. This can sometimes happen in frustrating cases — as in our search example where the structure of db may have to be known at compile time for the superstructure of the list of sub-databases to be available when constructing the annotation.

3.4 Example revisited

If we consider the same sequence of rewrite steps that were needed to reduce the annotation above, and re-interpret the term rewriting as term-graph rewriting, the computation graph is automatically updated to read as follows — where the **where** construct is used to label shared nodes:

```
search' keys db = replies
                  where
                  replies = merge splitresults
                  splitresults = splitresult₁ : splitresult₂ : [ ]
                  splitresult₁ = search keys db₁
                  splitresult₂ = search keys db₂
                  db₁ = hd splitdbs
                  db₂ = hd (tl splitdbs)
                  splitdbs = split 2 db
```

The annotation also reflects the sharing present:

```
Node splitresult₁ And Arc replies splitresult₁ And Node splitresult₂ And
Arc replies splitresult₂ And Node splitresult₁ And Arc keys splitresult₁ And
Node splitresult₂ And Arc keys splitresult₂ And Node keys And Node replies
```

Notice here that the elements of splitresults are the subject of the annotation, not the list splitresults itself. Thus, the list construction operations ":" which build splitresults are part of the process labelled replies.

4 Network extraction

The next phase is to take the program containing a normal form **moreover** annotation and replace the annotated block with an application of the special function procnet (described below). This gives an explicit representation of the process network to be created at run-time.

4.1 The procnet function

Procnet takes two parameters: a list of functions representing processes and a wiring list, and returns a function specifying the overall behaviour:

```
procnet :: [Process] → [Connection] → Process
```

Each function in the list specifies the input/output behaviour of a process: it takes as input a list of streams of messages, and delivers a single stream of messages as output (we actually return a singleton list of messages to accommodate multi-output processes):

```
type Process = [[Message]] → [[Message]]
```

The wiring list contains an element for each communications channel required:

```
type Connection = ((Int, Int), (Int, Int))
```

A connection ((i,j),(m,n)) indicates that output j of process i should be connected to input n of process m. Processes are indexed from one. This allows for a pseudo process, zero, that represents the rest of the world (which provides input to the network and consumes its output).

It is possible to specify procnet as a functional program. For ease of exposition it is simpler to say that it connects up outputs of processes to inputs of processes as specified by the wiring list.

4.2 Translation to procnet form

The process of removing the **moreover** is quite simple. We start by finding a label for each expression that is to become a separate process. The function *NodesOf* does this.

$NodesOf\,(\mathsf{Node\ e}) = \{\mathsf{e}\}$
$NodesOf\,(\mathsf{Arc\ e_1\ e_2}) = \emptyset$
$NodesOf\,(\mathsf{And\ annot_1\ annot_2}) = NodesOf\,\mathsf{annot_1} \cup NodesOf\,\mathsf{annot_2}$

In the final network there will be one process for each of these named expressions.

4.3 The process placement rule

The network extraction phase has to decide what is local to an expression and what is imported from another process. We must abstract, to the top level of the process, references to streams that are to be imported so that they can be provided at run-time. This means that what (in the initial program) was a stream now becomes a function, whose argument is a list of imported streams.

To decide what is imported we use the Process Placement Rule: A value is computed locally by a process unless it has explicitly been placed elsewhere by name.

Thus the streams imported by a process are those which appear free in its node expression, or in any expression referred to by it. If we treat the program as a term graph and trace all the expression tree we can locate all references to imported names and abstract them accordingly.

4.4 The implementation of network extraction

The extractor must walk the program graph as if it is a term graph, abstracting references to imported streams. This is very closely related to lambda-lifting. In our initial implementation we use only a simple abstraction scheme — we abstract variables and not maximally free expressions so we do not end up with a program that respects the laziness of the original.

When the program gets to the extractor it has been type checked and had pattern matching removed. We implement the annotation as a form of identity function of type "$\mathsf{a} \rightarrow \mathsf{Placement} \rightarrow \mathsf{a}$". We need to remove this and replace it by a call to procnet.

Firstly we construct the set of exported names using *NodesOf*. We then use a pair of mutually recursive functions, *rec_run* and *rec_abstr*, to traverse the program graph. *Rec_run* takes an expression and returns an environment listing what has to be imported into that expression. It calls *rec_abstr* to follow named references (so the graph is treated as a term graph).

Rec_abstr takes a name that references an expression and locates its definition. It calls *rec_run* on the defining expression and then uses the environment returned to form the expression into a function whose extra argument is a list of the streams that must be imported.

If *rec_run* finds a reference to an expression that has already been abstracted it can replace it with an application of the new function version of the expression applied to a list of arguments that represent the names that the function needs to import. It then recursively calls itself on the newly built application to ensure that the newly supplied arguments are imported into the calling expression.

All that is left to do is construct a list of functions (the abstracted versions of each of the node expressions) and a list of connection tuples.

4.4.1 Network IO

Process networks accommodate the stream based IO paradigm quite naturally. For this reason we have deviated from Haskell's IO specification and just provided a basic stream presentation of IO. We use a pseudo process (numbered 0) to act as the real world, supplying input and consuming output from the network. This is the only part of the program that can exist off the process network so we have to ensure that all processing is carried out on the network.

We can do this by applying a small program transformation to the input program before letting the network extractor loose. The result of this transformation is to move any computation not covered by the **moreover** annotation onto the network.

If the main program is the expression:

```
main xs = f (res moreover Node res)
          where
          res = g xs
```

then the application of f is not performed on the network. We can simply replace the main expression by this:

```
main xs = top moreover Node top And Node oldtop
          where
          top = f oldtop
          oldtop = g xs
```

Now all computation that is performed by the program is covered by the annotation.

4.5 Explicit process placement of search example

After the network extraction phase the parallel search example given earlier has the following form:

```
search' keys
  = ( procnet [replies', splitresult_1', splitresult_2']
              [((0,0), (2,0)), ((0,0), (3,0)), ((2,0), (1,0)),
               ((3,0), (1,1)), ((1,0), (0,0))] ) keys
    where
    splitresult_1' [keys] = search keys db_1
    splitresult_2' [keys] = search keys db_2
    db_1 = hd splitdbs
    db_2 = hd (tl splitdbs)
    splitdbs = split 2 db
    replies' [splitresult_1, splitresult_2] = merge (splitresults' [splitresult_1, splitresult_2])
    splitresults' [splitresult_1, splitresult_2] = splitresult_1 : splitresult_2 : [ ]
```

Procnet takes the output of the first process, which applies the function replies', as the computation's output. The computation's input stream keys is referred to in the wiring list as output zero of process zero, and is routed as input to the two sub-search processes which evaluate $\mathsf{splitresult}_1$ and $\mathsf{splitresult}_2$ respectively.

Because db, the database to be searched, is not a run-time parameter of the computation, it is built into the sub-search processes instead of being passed using a communications channel.

5 Load-time system

Once simplification has been completed we are left with a program that contains nothing but standard Haskell code. This can be compiled using conventional compiler technology — with the advantage that we are not compromising the sequential performance of the generated code by placing parallel synchronisation points in it. For our compiler we are using a system under development by Glaser, Hartel and Wild [GHW90] at Southampton University. This is an optimised sequential compiler that produces instrumented C code as it's output.

The complete C code is compiled for *each* PE using the appropriate native compiler. Our target is a Meiko Computing Surface, containing 32 Inmos T800 transputers. This is supported by a communications toolkit called "CSTools". It provides machine independent communication and process management primitives. It also allows the programmer to build static descriptions of process networks before the surface is loaded. The surface is then loaded using a "`cs_build()`" call. The critical restriction is that the network cannot be re-switched with reasonable expense once the program has started.

This leaves us with one thing left to do: implement procnet. To the compiler procnet looks like any other function.

5.1 Procnet implementation on the host

Procnet is first called on the host. It takes a list of functions that represent the work of each process and a wiring list describing how the processes are to be linked. It's job is to set up the network to start computing. To do this it sets up a CSTools description of the network, calls CSTools to load up the network, sends configuration information to each processor (the index of the function in the process list that that processor is to compute) and then starts up the network.

Once it has done the initialisation it must supply the network with input and consume it's output. This can be done in the same way as for a node in the network (see section 6.2).

5.2 Procnet implementation on the PEs

In the same way that procnet is called first on the host, it is called first on each PE (as all PEs run identical code). But the version of procnet on the network nodes is different. It receives configuration information from the host (its index into the process list) and sets up transports between it and the nodes it must communicate with. Once this is done it constructs a closure for each input transport that contains the transport identifier and a function *ReadListItem()* (which is called to collect the next input value). This *ReadListItem()* function, when called, reads an item and produces a cons cell whose head is the item just read and whose tail is the closure (ready to read the next item).

Next procnet sets up a suspension that forms the output of the node's computation (i.e. the application of the processes function to the closures that represent the input). The last step is to enter a loop that calls the sequential evaluator to compute the next output element, send it to the output transports and repeat.

6 Run-time system

Now the network has been loaded and started we can look at how inter-processor communication works.

6.1 Semantic considerations, and deadlock

We have to design the communication system to respect the semantics of the unannotated program. This means that the only time our parallel program should fail is when the sequential one would have. Take the definition of the stream of Fibonacci numbers:

```
fibs = 1 : 1 : map2 (+) fibs (tl fibs)
```

Notice that each element of the stream depends on the immediately preceeding element, and the one before that. If we modify the definition slightly:

```
fibs = 1 : map2 (+) fibs (tl fibs)
```

then each element after the first element depends on itself. In semantic terms, the result is simply $1 : \perp$, but if the program is considered as a model of a network of concurrent processes, then the effect models deadlock (see Wadge [Wad81])[1].

In communication, our major concern is avoiding deadlock when the unannotated program is defined. This point becomes particularly clear when we look at a node, A, whose output is sent to two nodes, B and C. We must ensure that B does not have to wait for C to be ready to receive before B itself can collect it's input value. In terms of Kahn's original work on modelling concurrent processes functionally, we need a non-blocking output with unbounded buffering [Kah74].

6.2 Stream output

Keeping these semantic concerns in mind, we can discuss the mechanisms used for communication in the system. In the general case a PE must communicate it's output with k other PEs, so it will have k output transports. The PE must:

- Evaluate a stream of messages one-by-one, and send them on the output transports.
- Send values on each transport as soon as the value has been computed and the transport is ready to receive.
- In particular, the PE must not wait for transport i to accept a value before trying to send it to transports j, k etc.

To achieve this end we have k non-preemtively scheduled concurrent processes, one for each output transport. Initially, each has a pointer to the suspension representing the node expression that all have to output. They then compete to evaluate the output expression and send the results down their transports. This means that as the output stream unfolds in the node's heap each of the output processes send more of it down their transport. The output process that is currently ahead of the others (in the evaluation) is the one that calls the evaluator. Buffering is therefore provided naturally by the heap.

[1]Although the semantic evaluation is identified with non-termination, most implementations can easily distinguish between deadlock and non-termination by noting that a needed expression is already being evaluated.

6.3 Stream input

The input side is more straightforward. The requirements are to translate incoming messages into elements of the appropriate list and avoid unnecessary blocking of senders. At the same time we should also avoid excessive buffering.

We have already seen how *ReadListItem()* is used to pull the next input value over the network (section 5.2). Unfortunately this simple scheme is inefficient as all the buffering happens at the source. A more sophisticated implementation would attempt to buffer at both ends and send several values at a time (to reduce network latency overheads).

6.4 Garbage collection

We employ the standard garbage collector used for the sequential implementation. The only modification is to extend the root set to include all the sender processes' states. Thus, the collection time is not increased compared to the sequential case, and furthermore each PE can perform collection at any convenient time independently of the others. In particular, collection can proceed in parallel without coordination.

Conclusion

The main achievement we claim is the development of a high-level notation to control parallel program execution, which allows distributed memory multicomputers to be used efficiently, without compromising the performance of the host functional language implementation in any respect.

We have described our first implementation of Caliban except for the simplification phase, which is under development. The compiler imposes quite severe restrictions on the form of programs it can accept — static networks of processes, each with a single output stream of a special type.

Future work

In the short term, we plan to implement the simplifier, use a coercion mechanism to avoid the need for the explicitly-tagged Message type, and allow for multiple distinct output streams on each processor. This would be expressed using a new placement annotation, Bundle, which requires a list of expressions to share the same PE; in fact, Node is a limited instance of this more primitive mechanism:

```
  Node a = Bundle [a];
```

Handling this adds significant complexity to the run-time system, as the sender processes must be able to enter the evaluator concurrently.

Another addition we will make quite soon is a parallel implementation of `map`, which dynamically allocates free PEs in a self-scheduled fashion. This facility is absent from the static implementation of Caliban we have described but is often useful. It can be embedded within the existing Caliban system, and may be extended to allow nested parallel `map` operations, thereby capturing divide-and-conquer computations.

In the longer term, we are investigating non-static process networks. There is a spectrum of possibilities, ranging from phased computations, in which the configuration is changed periodically, through to what is essentially just grain size and data placement control in a shared-memory parallel graph reduction implementation.

Related work

Hudak's Paralfl implementation [Hud86] allowed the programmer to specify the PE where a computation should occur, using a run time variable calculation. Strand [AI 88] and Concurrent Clean [vEPS90] offer similar mechanisms with higher-level features. Magee and Dulay achieve a similar result with their configuration language approach to imperative parallel programming [MD91]. Caliban is an attempt to capture the pragmatic value of these ideas, while retaining a uniform, declarative presentation which takes full advantage of the abstraction mechanisms of functional programming.

Acknowledgements

Our thanks are due to our collaborators Hugh Glaser, John Wild, Pieter Hartel on the FAST project at the University of Southampton, to Susan Eisenbach and David Harper at Imperial College, to Jim Cownie at Meiko, and to Diomidis Spinellis and his colleagues who wrote the prototype Haskell compiler whose front-end we are using, as an undergraduate project. The work was funded in part by the UK Science and Engineering Research Council and the Department of Trade and Industry, under grant number GR/F 35081. The equipment was funded under grant number GR/G 31079.

References

[AI 88] AI Limited. STRAND-88 language definition. Technical report, AI Limited, Greycaine Rd. Watford, Herts, UK, 1988.

[BvEG+87] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Term graph rewriting. 1987. In [dBNT87, pages 141–158].

[dBNT87] J.W. de Bakker, A.J. Nijman, and P.C. Treleaven, editors. PARLE, *Parallel Architectures and Languages Europe*, volume I. Springer Verlag, June 1987. LNCS 258.

[GHW90] Hugh Glaser, Pieter Hartel, and John Wild. A pragmatic approach to the analysis and compilation of lazy functional languages. Technical report CSTR 90-10, Department of Electronics and Computer Science, University of Southampton, 1990. (In Proc. of the Workshop on Parallel and Distributed Processing, Sofia, 1990. North-Holland 1991).

[Gol88] Benjamin F. Goldberg. *Multiprocessor Execution of Functional Programs.* Research report, Yale University Department of Computer Science, April 1988.

[Hud86] P. Hudak. Para-functional programming. *IEEE Computer*, pages 60–70, August 1986.

[Hug84] J. Hughes. Why functional programming matters. Report 16, Programming Methodology Group, University of Göteborg and Chalmers Institute of Technology, Sweden, November 1984.

[HWe90] P. Hudak and P. Wadler (editors). Report on the programming language Haskell, a non-strict purely functional language (Version 1.0). Technical Report YALEU/DCS/RR777, Yale University, Department of Computer Science, April 1990.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74*. North-Holland, 1974.

[Kel89] Paul H.J. Kelly. *Functional Programming for Loosely-coupled Multiprocessors.* Pitman/MIT Press, 1989.

[MD91] J. Magee and N. Dulay. A configuration approach to parallel programming. In *PARLE'91*, 1991. To appear.

[vEPS90] M.C.J.D. van Eekelen, M.J. Plasmeijer, and J.E.W. Smetsers. Parallel graph rewriting on loosely-coupled machine architectures. Technical report, Faculty of Computer Science and Mathematics, University of Nijmegen, February 1990.

[Wad81] W.W. Wadge. An extensional treatment of dataflow deadlock. *Theoretical Computer Science*, 13:3–15, 1981.

[ZC90] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1990.