Concurrent Clean

Eric Nöcker, Sjaak Smetsers, Marko van Eekelen, Rinus Plasmeijer
University of Nijmegen*

Abstract

Concurrent Clean is an experimental, lazy, higher-order parallel functional programming language based on term graph rewriting. An important difference with other languages is that in Clean graphs are manipulated and not terms. This can be used by the programmer to control communication and sharing of computation. Cyclic structures can be defined. Concurrent Clean furthermore allows to control the (parallel) order of evaluation to make efficient evaluation possible. With help of sequential annotations the default lazy evaluation can be locally changed into eager evaluation. The language enables the definition of partially strict data structures which make a whole new class of algorithms feasible in a functional language. A powerful and fast strictness analyser is incorporated in the system. The quality of the code generated by the Clean compiler has been greatly improved such that it is one of the best code generators for a lazy functional language. Two very powerful parallel annotations enable the programmer to define concurrent functional programs with arbitrary process topologies. Concurrent Clean is set up in such a way that the efficiency achieved for the sequential case can largely be maintained for a parallel implementation on loosely coupled parallel machine architectures.

1. Introduction

1.1. Historical context

Concurrent Clean (Eekelen et al. (1990)) is an experimental, lazy, higher-order functional programming language based on term graph rewriting (Barendregt, Eekelen, Glauert, Kennaway, Plasmeijer and Sleep (1987a)). The first work on Clean started in 1984 in the Dutch Parallel Reduction Machine project (Barendregt, van Eekelen, Plasmeijer, Hartel, Hertzberger and Vree (1987),Brus et al. (1987)) in which the feasibility of the realization of a parallel reduction machine was investigated. The Nijmegen research focussed on the fundamentals of graph reduction and its implementation on sequential and parallel architectures. The fundamental idea is that graph reduction should not be considered as merely an optimisation in the implementation of functional languages, but that graph reduction is a fundamental basis for any implementation and that graph reduction itself must be investigated and optimised. In this context together with the University of East-Anglia a more general non-functional computational model, Generalized Graph Rewriting Systems (GGRS's) has been designed (Barendregt, Eekelen,

*Faculty of Mathematics and Computer Science, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands E-mail: `clean@cs.kun.nl`

Glauert, Kennaway, Plasmeijer and Sleep (1987b)) of which the semantics and pragmatics currently are further investigated in the Esprit Basic Research Action"Semagrap". The Dactl-language used in the declarative UK-Flagship projects is based on GGRS's (Glauert et al. (1987)) as well as the jointly with the University of East-Anglia (UEA) defined language Lean (Barendregt, Eekelen, Glauert, Kennaway, Plasmeijer and Sleep (1987b),Barendregt et al. (1988)). Based on restricted GGRS's the functional graph rewriting language Clean (Brus et al. (1987)) was developed as an intermediate language for the compilation of functional languages. Implementations (compilers and interpreters) of Clean (Brus et al. (1987),Nöcker (1989),Smetsers (1989)) have been developed as well as a Miranda-to-Clean conversion program (Koopman and Nöcker (1988)). Concurrent Clean is partly developed as a part of the Esprit TIP-M Tropics project.

1.2. The language Concurrent Clean

In this paper the language Concurrent Clean is presented that extends the sequential language Clean to a concurrent language suited for efficient code generation for both sequential and parallel machine architectures. Concurrent Clean has many features in common with other lazy, higher-order functional languages, such as a Milner/Mycroft based polymorphic type system (including algebraic types, synonym types and abstract types). A key aspect of the language is that the object that is manipulated is a graph and not a term. Consequently, the programmer can explicitly control sharing of computation. For instance, cyclic data structures can be created. The most important aspect of Concurrent Clean discussed in this paper is the way in which the order of evaluation can be controlled. Lazy evaluation can be locally changed in eager evaluation. Eager evaluation has the advantage that it can be implemented considerably more efficiently than lazy evaluation. Even more speed-up can be achieved by changing sequential evaluation into parallel evaluation.

1.3. Changing lazy into eager evaluation

An important feature of the Concurrent Clean system is that strictness annotations are generated automatically by a strictness analyser. This analyser has been designed and implemented based on the concept of abstract reduction (Nöcker (1990)). The strictness analyser is an efficient as well as powerful analyser that can deal with arbitrary data structures and higher-order functions. To change the default lazy reduction order into eager, also the programmer can put strictness annotations in the function definition themselves or in their type definition (Smetsers (1989)). Furthermore, considerable efficiency improvements can be realized by defining a special kind of data types: partially strict data types (Nöcker and Smetsers (1990)) that enable composite data structures to be handled on the stack completely without any heap usage.

1.4. Changing sequential into parallel evaluation

In Concurrent Clean, the programmer can control the parallel evaluation of the functional program with help of two annotations (Eekelen et al. (1991)). The annotations enable the programmer to assign processes to parts of the graph in such a way that arbitrary, possibly cyclic, process topologies can be specified.

With the same two annotations the programmer can specify that, when communication takes place, a value has to be communicated or that the expression to compute the value has to be shipped. Communication between processes takes place implicitly on demand via the concept of lazy copying (Eekelen et al. (1991),Barendsen and Smetsers (1992)).

Concurrent Clean is designed for the evaluation on loosely coupled parallel machine architectures. As a special case multi-processing on a single processor can be expressed. Complicated parallel algorithms which can go far beyond divide-and-conquer like applications can be specified. The design of Concurrent Clean is such that the sequential optimisations mentioned above can still be applied in the parallel case. A local reservation/locking mechanism is required that introduces a neglectable overhead.

In this paper an overview is given of the main features of the language Concurrent Clean (Section 2). In more detail it is explained how the (parallel) reduction order is controlled (Section 3). The sequential (Section 4) and parallel (Section 5) implementation of Concurrent Clean is treated. Performance figures are given in Section 6.

2. Overview of the Language

In this section we briefly introduce the flavour of Concurrent Clean by showing how some well-known functional programs are written down in this formalism. The first example shows how the factorial function can be specified in Clean:

```
MODULE Fac;

IMPORT delta;

RULE
::      Fac INT     →     INT                   ;
        Fac 0       →     1                     |
        Fac n       →     *I n (Fac (–I n))     ;

::      Start       →     INT                   ;
        Start       →     Fac 20                ;
```

A Clean program is composed of modules. Modules are hierarchical. The top-most module is the main module. In the main module a Start rule should be declared of which the left-hand-side consists of the symbol Start and the right-hand side corresponds with the initial expression to be computed.

With the IMPORT statement all predefined functions (delta rules) and predefined types are imported. –I (integer decrement) and *I (integer multiplication) are such predefined functions defined on the basic type INT.

Rules starting with :: are either new type definitions or type specifications of rewrite rules. In the latter case, the type of the corresponding function is specified. In a Clean program all the rules for a certain function are called the alternatives for that function. It is required that all the alternatives are grouped together. The reader will have inferred that the rule alternatives of a function definition have a priority: they are applied in textual order.

```
MODULE Map;

||      Example of how to use higher order
||      functions in Concurrent Clean

FROM deltaI IMPORT *I;

RULE
::      Square INT          →     INT                        ;
        Square x            →     *I x x                     ;

::      Map (⇒ x y) [x]     →     [y]                        ;
        Map f [ ]           →     [ ]                        |
        Map f [a | b]       →     [f a | Map f b]            ;

::      Start               →     [INT]                      ;
        Start               →     Map Square [42,43,44]      ;
```

In Clean comments can be specified via preceding the comment with ||. This has to be done on every line in which a comment is given. Square brackets are used for denoting lists: [ ] is an empty list, [a,b,c] a list containing the three elements a, b and c, and [a | f] denotes a list consisting of a list f prefixed with an element a.

The example also shows that higher order functions can be used freely. There is no difference between the use of full and partial (curried) applications of functions. Types of higher order functions are specified using ⇒ (prefix notation) which corresponds to → (infix notation) in languages like Miranda [1].

The following example is a solution for the Hamming problem: it computes an ordered list of all numbers of the form $2^n 3^m$, with $n, m \geq 0$. Note that with the explicit nodeid x, defined in the right hand side, a cyclic graph is created that allows the use of computations already performed.

```
MODULE Ham;

FROM deltaI     IMPORT *I          ;
FROM Map        IMPORT Map         ;
FROM Merge      IMPORT Merge       ;

RULE
::      Ham     →     [INT]                                      ;
        Ham     →     x: [1 | Merge (Map (*I 2) x) (Map (*I 3) x)]  ;
```

2.1. Type System

Concurrent Clean is a strongly typed language. It is, however, not required to declare the types of functions explicitly: types are deduced by the compiler from the information in the program. The (polymorphic) type scheme that is used for this purpose is based on a combination of the well-known Milner (1978) and Mycroft (1984) schemes.

The predefined types in Concurrent Clean and examples of denotations and predefined functions are listed below:

[1] Miranda^{TM} is a trademark of Research Software Ltd.

Basic types:	INT, REAL, BOOL, CHAR, STRING, FILE
Examples of denotations:	2, 0.31415E1, TRUE, 'a', "monkey"
Predefined functions:	+I, <R, NOT, =C, SLICE, FOpen
List and tuple types:	[**T**], ($\mathbf{T}_1$, ... ,$\mathbf{T}_n$) for types **T** and $\mathbf{T}_i$
Denotations for lists and tuples:	[1,2,3,4], [ ], [2 | [ ]], (1,'?',FALSE)

Defining New Types

There are three mechanisms to introduce new types: *algebraic* type definitions, *synonym* type definitions and *abstract* type definitions.

Synonym types allow the user to define a new name for an already existing type. These types are specified by means of a type rule having exactly one alternative of which the right-hand side is a type instance.

A *type instance* is either a type variable or an acyclic graph that has a root symbol that is a *type symbol* of which all the arguments are type instances. A type symbol is either a basic type symbol or a user-defined type symbol.

An example of a synonym type definition:

```
TYPE
::    Stack x          →    [x]              ;
```

With the aid of algebraic types it is possible to introduce a new concrete data type based on free algebras. These types are specified by a type rule whereof each alternative has a right-hand side with a unique root symbol: the *constructor*. The constructor is said to be of that specific *type*. All the arguments of the constructor are type instances.

Below examples of algebraic type definitions are given. The types Nat and List are defined. The constructors Zero and Succ are said to be of type Nat, Cons and Nil of type List x.

```
TYPE
::    Nat      →    Zero                  |
      Nat      →    Succ Nat              ;

::    List x   →    Cons x (List x)       |
      List x   →    Nil                   ;
```

Abstract types offer the possibility of hiding the representation of a certain type. To distinguish an abstract type definition from an ordinary type definition a special kind of type block is provided called an ABSTYPE-block.

Example of abstract type definition in Clean:

```
ABSTYPE
::    Stack x    ;
```

Abstract type definitions are only allowed in definition modules (see the section on modules). In the implementation module the abstract type should either be a synonym type or an algebraic type. The realisation of the type is invisible for the outside world.

Typing Functions

Each rewrite rule can be typed explicitly by the programmer. This type specification must immediately precede the corresponding rewrite rule.

When typing partial functions one has to ensure that the function symbol itself can be used as a constructor by giving an appropriate algebraic type definition for it. An error is generated at run-time if this has not been indicated properly (note that in general it cannot be detected at compile-time whether a function is partial). First an example that leads to a run-time type error:

```
RULE
::  F INT    →  INT        ;
    F 0      →  0          ;

::  Start    →  INT        ;
    Start    →  F 1        ;
```

Although the Clean program is correctly typed, the function F applied in the start-rule cannot be matched and therefore F 1 will not yield the required type: INT. At run-time, an error is generated.

The second example shows how partial functions should be typed in order to avoid run-time errors:

```
TYPE
::  Num            →  Zero               |
    Num            →  Succ Num           |
    Num            →  Pred Num           ;

RULE
::  Succ Num       →  Num                ;
    Succ (Pred n)  →  n                  ;

::  Pred Num       →  Num                ;
    Pred (Succ n)  →  n                  ;

::  Start          →  Num                ;
    Start          →  Succ (Succ Zero)   ;
```

The graph Succ (Succ Zero) in the start rule will not match any rule. Still it is correct because the graph is indeed of the wanted type (i.e. Num). Notice that Succ and Pred are used both as functions and as constructors. As constructors they may appear in the right-hand side of type definitions and are of type Num. As functions they also yield type Num.

2.2. Modules

A Concurrent Clean program may be split in several modules that can be compiled separately. A Concurrent Clean program consists of *definition modules* and *implementation modules*. An implementation module contains type and rule definitions that can be *exported* to other modules via its definition module. The latter consists only of a set of type rules, possibly including strictness information, for exported types and for exported functions. Special definition modules, which are called *system modules*, indicate that the corresponding implementation module does not contain ordinary rewrite rules but (abstract) machine code instead. On demand the compiler will substitute the code of a function 'in-line' at the place where this function is called.

2.3. Input and Output

To achieve an efficient implementation of IO facilities in Concurrent Clean the type FILE has been predefined. Besides that, a number of basic operations on files can be imported from a predefined module called deltaIO. This module contains functions to create files, to read characters or strings from files, to write characters or strings to files and to re-open write-files for reading.

The efficiency of the IO functions is obtained by implementing FILE's not as (lazy) lists of characters but by using strict tuples. This allows the Concurrent Clean compiler to generate code for these IO functions wherein a fast call by value like mechanism of parameter passing and returning results is used.

3. Controlling Reduction Order

3.1. Graph Rewriting

A Clean program basically consists of a number of graph rewrite rules which specify how a (program) graph has to be rewritten. The program graph, which initially consists of a single Start node, is rewritten according to these rules. The part of the graph that matches the pattern of a certain rewrite rule is called a *redex*. A *rewrite* of a redex consists of replacing the redex in the graph by an instance of the right-hand side of the corresponding rewrite rule.

3.2. Reduction Strategies

A reduction strategy repeatedly determines which redex is going to be reduced next. The strategy of Concurrent Clean is the so-called *functional strategy*. Reducing graphs according to this strategy resembles very much the way execution proceeds in many other lazy functional languages: if there are several rewrite rules for a particular function, the rules are tried in textual order; patterns are tested from left to right; evaluation of arguments is forced when it is tried to match an actual argument against a non-variable part in the pattern.

In Concurrent Clean the functional strategy may locally be influenced by the use of annotations. When this strategy encounters an annotation it changes its default reduction order which will influence the way a result is achieved. Changing the order is in particular important if one wants to optimise the time and space behaviour of the reduction process.

Currently, two kinds of annotations are possible:

- *strict annotations* to locally change lazy evaluation into *eager evaluation*;
- *process annotations* to define *interleaved evaluation* on the same or *parallel evaluation* on another processor.

3.3. Sequential Annotations

The sequential flow of control can be influenced by means of strict annotations. If a strict annotation is encountered, the evaluation of the indicated subgraph is forced. This forced evaluation will also follow the functional strategy yielding a root normal form. After the forced evaluation has delivered the root normal form, the reduction process continues with the ordinary reduction order following the functional strategy. So, annotations let the reduction strategy deviate from the default functional evaluation order making the evaluation order partially eager instead of lazy.

We distinguish two kinds of strict annotations, namely *global* and *local* strict annotations

Global Strict Annotations

The strict annotations in a type specification are called global because they change the reduction order for *all* applications of a particular function. Annotations in a type specification of a certain function are allowed to be placed before the type specification of either an argument on the left-hand side or an argument of a tuple type appearing in a *strict context*. A tuple type is in a strict context if it has been supplied with a (valid) strict annotation itself or if it appears as the root node on the right-hand side of the type rule. Intuitively, such a strict annotation indicates that the corresponding argument is always reduced to root normal form before the corresponding rule is applied.

Example of global strict annotation in type rules:

```
::  IF !BOOL x x         →  x     ;
    IF TRUE then else    →  then  |
    IF FALSE then else   →  else  ;
```

Strict annotations may also be used in tuple types appearing in a type synonym definitions. The meaning of these annotated synonym types can be explained with the aid of a simple program transformation with which all occurrences of these synonym types are replaced by their right-hand sides (of course, annotations included). These annotated type definitions are a special case of the more general *partially strict data types* which are treated later on in this section.

Local Strict Annotations

Strict annotations in rewrite rules are called *local*. They change only the order of evaluation for a *specific* function application. These annotations appear in the right-hand side of rewrite rules.

Before the evaluation continues after applying a rewrite rule, all strict annotated nodes of the right-hand side of the applied rewrite rule are evaluated. Strict annotations in rewrite rules can be placed anywhere on the right-hand side.

Example of strict annotations on the right-hand side:

```
F x y          →    IF x !y (! ++I y)     ;
```

In this particular application of IF it is clear that a common part of the then part and else part can safely be reduced.

Partially Strict Data Types

Partially strict data types (Nöcker and Smetsers (1990)) are obtained by supplying the type definitions or type specifications of functions with additional (global) strictness information. In a type definition this strictness information specifies for each individual part of an instance of such a type whether this part should be evaluated or not (the so called evaluation context of that part). In a type specification of a function the strictness information determines the evaluation contexts of both the parameters and the result. The only partially strict data types that have been implemented in Concurrent Clean are the partially strict tuples (these types were already mentioned in the section on global strict annotations). An example of the use partially strict tuples is the following definition of a complex number:

```
TYPE
::    Complex                 →    (!REAL,!REAL)          ;

RULE
::    +C !Complex !Complex    →    Complex                ;
      +C (r1,i1) (r2,i2)      →    (+R r1 r2,+R i1 i2)    ;
```

3.4. Parallel Annotations

The parallel flow of control can be influenced by means of process annotations. Currently, only *local process annotations* can be specified in the right-hand side of rewrite rules.

If a process annotation is encountered, the evaluation of the indicated subgraph is forced as with a strict annotation, following the functional strategy until a root normal form is reached. The important difference with strict annotations is that with process annotations new reduction processes are created that perform the evaluation. These new reduction processes can run interleaved or in parallel with the original reduction process. The original process continues with the evaluation in the ordinary reduction order independently.

Creating parallel processes

The {P} annotation (P for parallel) creates a new graph, which is a copy of the annotated subgraph, on a remote processor together with a parallel reduction process (a *reducer*) which reduces this new graph to root normal form.

Creating interleaved processes

The {I} annotation (I for interleaved) creates a new internal process on the annotated subgraph. This new internal reducer reduces the corresponding subgraph interleaved with the other processes of this processor (so no copy is made).

Communication Channels

Communication takes place when the initial graph that is going to be reduced in parallel has to be sent to another processor or when the result of such a parallel reduction is needed by another reducer.

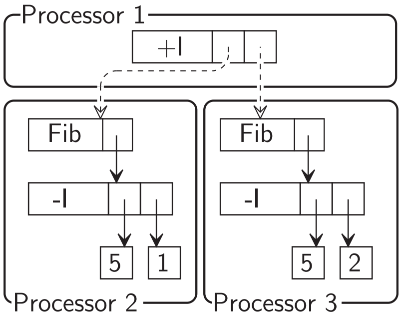
Communication involves the copying of graphs. In Concurrent Clean the concept of *lazy copying* is used (Eekelen et al. (1991),Barendsen and Smetsers (1992)). When during the copying a subgraph is encountered that is already being reduced by another reduction process, this subgraph is not copied at that moment. The copying is deferred until the other reducer has finished the reduction of this graph. The fact that the copying was stopped temporarily is administered with the aid of a special arc, a so-called (*communication*) *channel*, that interconnects the new copy with the subgraph that is currently reduced. The continuation of the copying is triggered when the result of the graph to which a channel refers is needed. Besides creating channels implicitly via copying there is another way whereby channels come into existence: the initial subgraph of a new parallel reduction process is also connected to the original graph via a channel. Note that the above-mentioned method of process creation and communication implies that the only interconnections between graphs residing on different processors are channels.

Divide and Conquer Parallelism

In the following example it is shown how divide-and-conquer parallelism can be specified in Concurrent Clean:

```
Fib 0   →   1                        |
Fib 1   →   1                        |
Fib n   →   +I left right,
            left: {P} Fib (-I n 1),
            right: {P} Fib (-I n 2)  ;
```

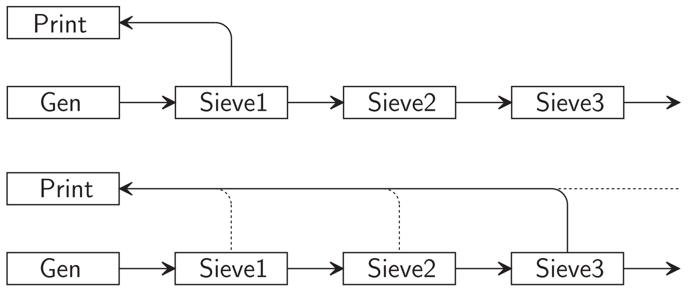
The {P} annotations specify that both calls of Fib can be evaluated in parallel. The root of the graph on which a process is started, is built on another processing element with copies of subgraphs as arguments. The father reducer is waiting for the results. A copy of a result is made when a subgraph left or right is in root normal form. The picture below illustrates a possible processor structure after one reduction of Fib 5:



Parallel sieving

The sieve of Eratosthenes is a classical algorithm for generating prime numbers. A pipeline of Sieve processes is created. Those Sieves hold the prime numbers in ascending order, one in each Sieve. Each Sieve accepts a stream of integers as its input. Those integers are not divisible by any of the foregoing primes in the pipeline. If an incoming integer is not divisible by the local prime as well, it is sent to the next Sieve. A newly created Sieve accepts the first incoming integer as its own prime and outputs this prime and the channel of the next Sieve to a printing process. After that it starts sieving. A process called Gen sends a stream of integers greater than one to the first Sieve. The combination of process annotations and communication via copying provide that the intended behaviour is achieved. Processes are connected to each other by channels through which data is communicated in a demand driven way.

This can be represented in a picture as below (all arrows indicate flow of data on channels). Sieve1 holds 2 as its own prime, Sieve2 holds 3, Sieve3 holds 5, and so on. The printing process one by one receives the channel identifications from these sieves and collects the corresponding primes. Seen through the time this can be illustrated as follows (all arrows indicate flow of data on channels):



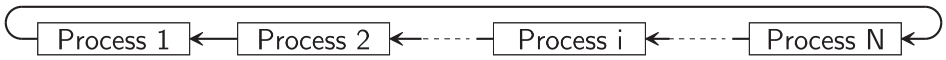
The Sieve program:

```
Start               →    Print s,
                         s: {P} Sieve g,
                         g: {P} Gen 2                ;

Sieve [pr | stream] →    [pr | s],
                         s: {P} Sieve f,
                         f: {I} Filter stream pr     ;

Gen n               →    [n | rest],
                         rest: {I} Gen {!} (++I n)   ;

Filter [f | r] pr   →    IF    (=I (MOD f pr) 0)
                               (Filter r pr)
                               (NewFilter f r pr)    ;

NewFilter f r pr    →    [f | rest],
                         rest: {I} Filter r pr       ;
```

Arbitrary Process Structures

It is beyond the scope of this paper to treat the expressive power of Concurrent Clean very extensively. At this point we only want to claim that it is possible to specify any arbitrary process structure in a Concurrent Clean program. To illustrate this we give an example that shows how a cyclic process structure, i.e. a number of parallel reducers that are mutual dependent, can be created. It is extracted from quite a large program that implements Warshalls solution for the shortest path problem (Eekelen (1988)).

First the intended reducer topology is given in a picture:



This reducer structure can directly be specified in the following way:

```
Start                  →   last:CreateProcs NrOfProcs last     ;

CreateProcs 1 left     →   Process 1 left                      |
CreateProcs pid left   →   CreateProcs (–I pid) new,
                           new: {P} Process pid left           ;
```

CreateProcs is responsible for the generation of all the parallel reducers. This process, which will finally become the first reducer, has initially a reference to itself in order to make it possible to expand it to a cycle of reducers. Each reducer is connected to the next one, i.e. the one with the next pid number, by means of a channel. During the creation of the processes this channel is passed as a parameter called left.

4. Sequential Implementation

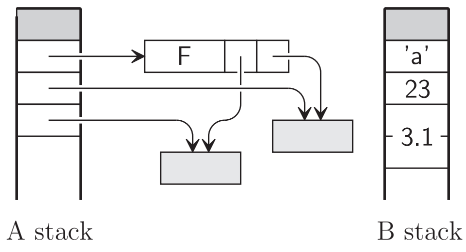
Both sequential and parallel implementations of Concurrent Clean are based on the abstract ABC machine. A Concurrent Clean program is compiled to code for this abstract machine. In this way the Concurrent Clean compilation is largely machine independent. Testing the implementations and reasoning about them becomes much easier. There are two ways in which this code can be executed. First, the ABC code can be interpreted. Second, it can be compiled to code for some concrete machine. The abstract machine can be implemented on various machines relatively easy. In this section we will outline the basic aspects of the ABC machine. The ABC machine resembles advanced G-machine like architectures (Johnsson (1987),(Peyton Jones and Salkild (1989)). The Concurrent Clean compiler exploits all possibilities of the machine. This is discussed in section 4.2. Lastly, we treat how the ABC machine can be implemented on a real machine. More detailed information on these aspects can be found in Smetsers (1989),Koopman et al. (1990) and Groningen (1990).

4.1. The abstract ABC machine

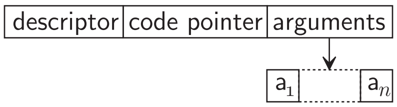
As mentioned before, the abstract ABC machine is similar to G-machine like architectures: it is a stack based graph reduction machine. The main parts of interest are the three stacks (Address, Basic value and Control stack) and the heap.

The C stack is used for storing addresses. The other two stacks are used for evaluating or building expressions, and for passing arguments to functions or returning results from functions. The A stack contains references to nodes in the heap, whereas the B stack contains values of basic types, such as integers or reals. Thus, basic values can be represented in two ways: as node in the heap or as an item on the B stack. Note that a B stack item can occupy more entries, for example, a Real value needs two entries.

Example:



Graphs are stored in the heap. So, the heap contains a collection of nodes. Generally speaking, a node of a Clean graph consists of a symbol with a certain number of arguments. Representing nodes as variable sized object causes problems with updating: the new node does not need to fit in the old one. This can be solved by introducing indirection nodes, but this will slow down the access on the contents of a node. In the ABC machine we have chosen to split a node in a fixed and a variable sized part. The fixed size part contains a representation of the symbol (called the *descriptor*), a code pointer and a pointer to a variable sized part.



The descriptor is a representation of a Clean symbol. Normally it is an index or pointer in a descriptor table. The descriptor is used for pattern matching and for printing.

The code pointer points to code with which the node can be evaluated. During reduction this code pointer may change. For example, after entering the node for evaluation a pointer to an error routine is stored. If the node is entered again (indicating a non-terminating reduction) this code will be executed. If a node is updated with a head normal form value, the code pointer points to special code just containing a return instruction. In the variable sized part the arguments of the node are stored. This means that the arguments always have to be fetched via an extra indirection. On the other hand, updating a node is simple: update the fixed part, and allocate space for the arguments.

Except nodes containing a Clean symbol with the right number of arguments also other kinds of nodes are possible. For such nodes special things are done.

For nodes containing a basic value, e.g. an integer, the descriptor does not represent the Clean symbol (that would be the integer value itself). Instead, all integers share the same descriptor (e.g. INT). The integer value itself is stored in the pointer part. For basic values that do not fit in the fixed part of a node (e.g. strings) a pointer to the value (for which space has to be allocated) is stored. Since basic nodes are always in normal form, they all contain the head normal form code pointer.

In Concurrent Clean, symbols can be applied on too few arguments. Such a partial application can be represented as a spine of applications. In practice, a better way is to build partial nodes, i.e. nodes with a partially filled argument part. Such nodes are built as standard nodes, but with special descriptors. So, for each Clean symbol of arity n, n+1 descriptors are defined. Mostly, the ABC machine sees no difference between such partial nodes and standard nodes. However, if a partial node is applied to another node, a new node with a new number of arguments has to be created.

4.2. The Concurrent Clean compiler

The main task of the Concurrent Clean compiler is to generate efficient ABC code. The syntax of Concurrent Clean is rather simple: no complex transformations like lambda lifting or the conversion of ZF-expressions are necessary. Many standard optimisation techniques are implemented: tail recursion removal, avoiding unnecessary evaluation calls, and so on. In the following, we will emphasize only those parts of the compiler that differ from other well-known implementations.

Conceptually, graph reduction is done in the heap: if a node has to be rewritten a new graph is built which will replace the original node. Unfortunately, this scheme will not give efficient code. The goal of the compiler is to generate code in which graph building is omitted as much as possible. For generating such efficient code type and strictness information is necessary. Type information can be fully derived by the type inference mechanism. Strictness information can be given by the programmer, or can be derived by a strictness analyser.

In general, deriving strictness information is very difficult. However, some help from the programmer normally will lead to more information. Certainly annotating data types can lead to much more efficient code.

Strictness Analysis

The strictness analyser in the Concurrent Clean compiler is based on *abstract reduction* (Nöcker (1990)). In abstract reduction a domain of sets of values is defined. Reduction in this domain means reduction of sets. Because this domain of sets is not finite, fixed points techniques are not applicable. Problems due to recursive functions are solved with a technique called *reduction path analysis.* With this method also other kinds of strictness can be derived, for example, strictness properties for functions over lists. (however, such information is not used by the Concurrent Clean compiler). It appears that this analyser can find much information. The analysis itself is quite fast. Consider the functions:

```
Append [ ] y          →   y                     |
Append [a | r] y      →   [a | Append r y]      ;

Foldr op r [ ]        →   r                     |
Foldr op r [a | x]    →   op a (Foldr op r x)   ;

Catenate l            →   Foldr Append [ ] l    ;
```

With strictness analysis based on abstract interpretation for the function Catenate a fixed point in a rather complex domain has to be determined. With abstract reduction the right information is found quite easily (see Nöcker (1990) for the analysis).

Nodes in a strict context

There are two ways in which the compiler uses strictness information. First, nodes in a strict context normally do not need to be built. Instead, a call to the code belonging to the function is generated:

```
F x    →    +I a b,
            a: IF cond 3 b,
            b: IF cond a 4,
            cond: P x ;
```

As can be seen easily, the node cond is in a strict context. In this case a direct call to P can be generated. However, despite the fact that nodes a and b are in a strict context, nodes for them have to be build because they are on a cycle.

Passing parameters and returning results

The second way in which strictness and type information is used is in passing values as parameters or as results. Values are passed via the A and B stack. The type of the function determines how this is done:

```
::    F INT ! (! INT,! [CHAR])    →    ( INT,! CHAR);
```

The function F is a function requiring two arguments. The first one, is a non-strict integer. This value is passed via the A stack. The second argument is a strict tuple. Both elements of this tuple have to be reduced to head normal form before calling F. The integer has to be passed via the B stack, whereas the character list is passed via the A stack. For the result value similar things have to be done: a (strict) tuple is returned of which the first element, a non-strict integer, will be passed via the A stack, and the second, a strict character, will be returned via the B stack.

If a value is not in the state in which it is needed for a function call, a conversion has to be done. In the case of tuples, such a conversion (which is called a *coercion*) can be quite complex.

Entry points

The above calling convention is applicable only if nodes appear in a strict context. However, there are three other ways in which a function can be called.

Firstly, a function application might have been appeared in a non-strict context. In this case a node has been built. If this node is evaluated, first a conversion (in fact a coercion) has to be done before the strict code can be executed: arguments have to be fetched from the heap. If necessary, they have to be evaluated or, in the case of strict tuples, unpacked.

The second way in which a function can be called is if a partial application has been built. After that some applications have delivered the remaining arguments, a similar transformation has to be done. Lastly, also special things have to be done for exported functions. The exported type determines the calling convention outside the module. However, inside the module another calling convention can be more efficiently. This is the case if abstract types are exported (hiding the internal representation), or if the strictness analyser finds more information than is exported. For both cases an additional entry point is needed. This 'external_strict' entry does some conversions according to the extra strictness information and continues with the internal strict entry.

So, in general the layout of the code of a function is as follows:

```
apply_entry:
        get arguments
        jump convert_code
lazy_entry:
        get arguments
convert_code:
        convert strict args
        jump_subroutine to strict_entry
        update node
        return
external_strict_entry:
        convert strict args
strict_entry:
        ...
```

For some functions (e.g. many predefined functions like +I etc.) special things are done. A call (in a strict context) to an addition would be unnecessary expensive. Instead, the addition code itself will be substituted directly. This is done by .inline directives that are inserted in the strict code part. Such inline code is only searched for if the predefined function was imported from a SYSTEM module. Note that the compiler itself knows nothing about such functions. In this way new basic functions can easily be added. Even functions for which complex code has to be inserted can be expanded inline in this way.

4.3. Realisation on a concrete machine

Basic Aspects

There are two ways of implementing the ABC machine on sequential hardware: by means of an ABC code interpreter and by means of a code generator that compiles ABC-code into target machine code. The section gives a short description of the code generator for the MOTOROLA 680x0 processors. The interpreter is treated in the section on the current status of our research.

Code generation for an M68k processor

A straightforward way of generating concrete machine code is by means of macro expansion: each ABC instruction is considered as a macro application that is substituted by a sequence of M68k instructions. However, the quality of the generated M68k code is mainly determined by the way the registers of this processor are utilised. Since the ABC machine does not contain abstract registers it will be evident that the resulting code is far from optimal. Therefore, the current ABC to M68k code generator uses a more intelligent way of generating code than just performing macro expansion. An ABC program is subdivided into *basic blocks* (i.e. sequences of ABC instructions that do not contain any label definitions or jump instructions). The code generator considers each basic block as a specification of how the initial state of the ABC machine (which is determined by the contents of the stacks and the graph store) at the start of the basic block has be converted into the final state at the end of the block. Now the tasks of the code generator becomes to implement such state transitions as efficient as possible, in all likelihood, by using registers. Note that, in contrast with the macro expansion mechanism, the relation between original ABC code and generated M68k code may be difficult to detect.

Besides using registers for computing intermediate results inside the basic blocks, registers are also used for parameter passing and returning results between basic blocks. As an example we give both the ABC code and M68k code generated for the factorial function that has been defined earlier:

ABC-code:	M68k-code:
Fac.1:	Fac.1:
eql_b +0 0	cmp #0,d0
jmp_true lab	bne Fac.2
jmp sFac.2	
lab:	
pop_b 1	
pushI +1	move #1,d0
rtn	rts
Fac.2:	Fac.2:
push_b 0	move d0,(a4)+
decI	sub #1,d0
jsr Fac.1	jsr Fac.1
push_b 1	
update_b 1 2	
update_b 0 1	
pop_b 1	
mulI	mul -(a4),d0
rtn	rts

The previous example clearly shows that the main task of the Concurrent Clean to ABC-code compiler is to define some order of evaluation in which the B-stack is used if possible. It does not try to optimise the stack manipulations, for instance by avoiding redundant move operations. Such optimisations are done by the ABC to M68k code generator.

5. Parallel Implementation

Also the parallel implementation is based on the ABC machine. In this section we will present the parallel ABC machine, and its implementation aspects.

The basic assumption we make for this parallel machine is that each processor has its own local memory. On each processor a number of sequential ABC machines can be running. For each new process, created by a {P} or {I} annotation, a new sequential ABC machine (a *reducer*) is started. Reducers have their own stacks. Reducers on the same processor share the heap of that processor.

5.1. The reservation/locking mechanism

Because several reducers on one processor can share subgraphs, some reservation mechanism is necessary. In the parallel ABC machine this is done as follows.

A reducer evaluates a node by executing the code pointed to by the code pointer of that node. The first this code does is changing the code field of the node. The new code pointer points to a piece of code with which other reducers that will try to evaluate this node will be suspended:

```
_reserve:
        set_wait 0
        suspend
        rtn
```

If a reducer executes this code sequence it puts itself (by the set_wait instruction) into the waiting list of the node it wanted to reduce. Thereafter, it suspends itself with the suspend instruction.

After some time the node will be updated by the first reducer (note that nodes are updated only with head normal forms). Then also the reducers in the waiting list will be released. They all execute the return (rtn) instruction and continue as if they had reduced the node themselves.

In first instance, it seems as if a waiting list will enlarge the fixed size part of all nodes: each node must have enough room to store a pointer to such a list. However, a node with a waiting list is under reduction and no information of this node is needed anymore. Therefore, in a concrete implementation other fields of the node can be misused.

5.2. Communication

There are two moments at which a graph has to be shipped to another processor. First, with the {P} annotation a remote reducer has to be started. The graph this reducer has to evaluate has to be copied to the processor on which the reducer will be started. The second case occurs if a result of a reduction is needed on another processor.

These forms of graph copying are basically the same. Copying a graph is not straightforward, since its structure has to be preserved. So, the copying algorithm has to take account of sharing and cycles. Also special action is needed if reserved nodes or nodes on which a reducer will be started (by an {I} or {P} annotation) risk to be copied. Reserved nodes can be recognised by the code pointer (or, alternatively, a flag might have been set). For nodes on which a reducer will be started a special node, called a *Defer* node, is inserted. In both cases simple copying of these nodes would mean duplication of work. Instead special nodes are created: *channel* nodes. Such a channel node is also created in the case of the {P} annotation: it points to the graph that will be reduced by the new remote reducer. So, a channel node can be considered as a node containing a pointer to a remote graph. It has a special code pointer:

```
_channel_code:
        set_entry _reserve 0
        send_request 0
        suspend
        rtn
```

If such a node is evaluated a request will be sent to another processor (by the send_request instruction). Then the reducer will suspend itself. As soon as the requested graph is in head normal form it will be sent. The channel node will be updated with this graph. Note that a request is sent only once: the code pointer is set to the reserve code, so other reducers will be suspended immediately. If a channel node is reduced, it is needed. Thus a request is sent only if the channel node is needed. Lastly, we note that also channel nodes can be copied. The result will be a copy of the channel node.

6. Results

6.1. Current Status

Currently, the Concurrent Clean to ABC compiler has fully been implemented on various machines. It includes all aspects mentioned earlier and it compiles quite quickly. On a SUN3/280 it compiles roughly 150 lines of Concurrent Clean code per second. This is without strictness analysis. With strictness analysis compilation time approximately doubles.

For the ABC machine both a simulator and several code generators exist. The simulator is used for testing both sequential and parallel versions of the ABC machine. For the parallel part the simulator has some global knowledge of a real run time system of a parallel machine. In particular, it includes a parallel garbage collector, and a stack reallocation mechanism.

At this moment several versions of an ABC code to machine code compiler are available. The best one generates code for the MC68020 type of processor, and has been implemented both on the MacintoshII as well as on a SUN3. Also for the Transputer a code generator exists. The last one is a preliminary version of a code generator for a parallel machine. In the future this code generator will be extended with the same optimisation techniques as the other ones.

6.2. Sequential

We compared the implementation of our system with implementations of Lml, Hope and C on the SUN3 (with a MC68020, 25Mhz processor). The Lml system is considered as a standard implementation of a lazy functional language (notice that we do not present figures for Miranda: most of the benchmarks below do not terminate within reasonable time). The Hope system is an example of a fast implementation of a strict functional language. The imperative languages are represented by C. We note that, if possible, C has been used in an imperative way (i.e. using iteration instead of recursion). The following implementations of these languages were used:

Lml	The Chalmers Lazy ML compiler, version 0.99.2, (90/08/20) (Augustsson and Johnsson (1989)).
Hope	The Hope$^+$ compiler, release 3.2.1, August 1989 (Burstall et al. (1980)).
C	The gnu C compiler, version 1.36 (which generally gives faster code than the standard C compiler).

The following test programs were used:

nfib	the well known nfib program with argument 30.
tak	the Takeuchi function, called with (tak 24 16 8).
sieve	a program which generates the first 10000 primes, using quite an optimal version of the sieve of Eratosthenes (outputs only the last one).
queens	counts all solutions for the (10) queens problem.

reverse	a program which reverses a list of 3000 elements 3000 times.
twice	four times the twice on the increment function.
revtwice	four times the twice of the reverse of a list of 30 elements.
rnfib	again the nfib program, but now working on real numbers, with argument 26.
fastfourier	the fast fourier algorithm, on an array of 8K complex numbers. In the Concurrent Clean program a complex number is defined as a strict tuple of two reals.

	Clean	Lml	Clean (u!)	Hope	C	Clean (-!)
nfib	4.5	25	4.5	5.4	11	30
tak	4.9	40	4.9	7.2	11	36
sieve	8.1	25	6.8	9.1	4.5	12
queens	28	62	14	16	4.1	45
reverse	64	108	50	65	–	51
twice	1.7	SF	0.5	0.3	–	1.7
revtwice	27	OH	9	12	–	39
rnfib	11	26	11	33	19	19
fastfourier	34	–	19	–	9.0	–

Table 6.1 Performance Overview (All times in seconds cpu time)

The following notes have to be made:

- The Lml versions of twice and revtwice resulted in run-time errors for these values (SF and OH stand for 'segmentation fault' and 'out of heap' respectively).
- The reverse and twice programs make no sense in the C context. The sieve and fast fourier programs are iterative versions. The other ones are inherently recursive.
- Computing the fast fourier with the other functional languages is impossible: they all would run out of heap space.
- The times needed to generate an executable for the example programs vary widely. On an average, the Concurrent Clean implementation consumes about 3.5 seconds cpu time, the Lml system needs 6 seconds and the Hope system even 15 seconds.

The first two columns of the table compare a standard compilation of Concurrent Clean programs with Lml. The default reduction strategy is lazy, but strictness information is added automatically by the strictness analyser. It is obvious that in all cases Concurrent Clean outruns Lml.

The next two columns present a comparison between user annotated Clean and Hope. User annotations are inserted at some places that are not indicated by the strictness analyser. Some of these annotations can be found automatically by a clever analysis (but not by strictness analysis), as is the case for the sieve and the queens programs. The annotations for the fast fourier (in the type definition of the complex number) have to be added by the programmer. Again, Concurrent Clean produces in almost all the cases the fastest code although the differences are not that great anymore. The only case in which Hope is faster is the twice example. This is mainly because Hope uses a smart integer representation. This is indicated by the revtwice program, which also tests the implementation of higher order functions but avoids the use of integers.

The recursive programs written in C appear to be slower than the ones written in Concurrent Clean. However, the iterative versions of the examples written in C are faster. But, in comparison with the past, the difference between execution times of on the one hand the functional languages and on the other hand the imperative languages has significantly decreased.

The last two rows of the table are measurements for real arithmetic. In fact, they show that of the functional languages only Concurrent Clean supports reals seriously.

Finally, the last column gives execution times for Concurrent Clean programs for which no annotations were added, neither automatically by the strictness analyser, nor by the programmer himself. From these figures we can conclude that in general strictness annotations increase the efficiency. The largest gain is achieved in programs which largely manipulate objects of basic types as is the case with tak and fast fourier.

6.3. Parallel

Partly funded by the ESPRIT Parallel Computer Action and the Dutch Neural Network Project, recently a beginning has been made with the implementation of Concurrent Clean on a Transputer system composed of 64 Transputers. Currently, this implementation supports only multi-processing on a single Transputer. Therefore, it is not yet possible to present performance figures of executions on a real parallel machine. However, with the PABC simulator a number of preliminary observations have been made.

The main results concern the kinds of parallelism which are possible, and how the parallel annotations influence this.

The process annotations are very powerful: it appears that many kinds of parallelism can be created. Also, it appears that the optimisations of the sequential code can be used in the parallel programs. The main problem in here is to assure that the grain size of the tasks is big enough.

The main disadvantage is that often very many reducers are needed to achieve a certain behaviour (for instance, each channel requires a reducer serving it). Also, the process annotations have to be used very carefully. Sometimes they have to be combined with local strictness annotation to provide that processes are created at the moment they are wanted. Some programs tend to behave sequential or create too many reducers if annotations are used wrongly.

7. Future work

The efficiency of the sequential code can be further improved by adding a special so-called "application depended strictness analysis" to the system. Such an analysis tries to determine whether eager evaluation of arguments for a certain application is safe because for this specific application it is known that these arguments will be evaluated (inspite of the fact that the applied function is not known to be strict in these arguments for the general case). Program transformations will be investigated that yield larger basic blocks of ABC code such that an optimal use of the new code generator is made.

We hope to demonstrate in the near future that real speed-ups can be achieved on a parallel architecture such as a Transputer system (Kesseler (1990)). At UEA already some promising results have been obtained with a previous version of our Clean system (McBurney and Sleep (1990)).

Furthermore, the presented annotations will be extended in order to enable the fine tuning of load balancing on a parallel machine.

On a higher level of abstraction new annotations are investigated to make parallel functional programming more user friendly (Eekelen and Plasmeijer (1990)).

8. Conclusions

The language Concurrent Clean is a lazy, higher-order functional graph rewriting language with as special feature that the sequential and parallel reduction order can be controlled in a general way. In Concurrent Clean arbitrary, dynamically changing process topologies can be specified. Parallel evaluation and communication can be controlled by the programmer.

There are several optimisations incorporated in the compiler such that, after a reasonable compilation time, very efficient execution is obtained for the sequentially evaluated parts of the code. The differences in speed between functional programs written in Concurrent Clean and programs imperatively written in a language like C are now becoming acceptable. Most optimisations are still applicable when code is generated for parallel environments.

The expressive power of the concurrency primitives available in Concurrent Clean makes it possible that a new class of parallel algorithms can be expressed adequately in a functional language.

Simulations have shown that the speed obtained for sequential machines can be inherited for parallel architectures such that efficient, parallel functional programming will be possible.

References

Augustsson, L. and T. Johnsson (1989). The chalmers lazy-ml compiler, *The Computer Journal*.

Barendregt, H.P., M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer and M.R. Sleep (1987a). Term graph reduction, *Proc. of Parallel Architectures and Languages Europe (PARLE)*, Eindhoven, The Netherlands, Springer Verlag, LNCS 259 II, pp. 141–158.

Barendregt, H.P., M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer and M.R. Sleep (1987b). Towards an intermediate language based on graph rewriting, *Proc. of Parallel Architectures and Languages Europe (PARLE)*, Eindhoven, The Netherlands, Springer Verlag, LNCS 259 II, pp. 159–175.

Barendregt, H.P., M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer and M.R. Sleep (1988). Towards an intermediate language based on graph rewriting, *selected papers of the conference on Parallel Architectures and Languages Europe (PARLE).*

Barendregt, H.P., M.C.J.D. van Eekelen, M.J. Plasmeijer, P.H. Hartel, L.O. Hertzberger and W.G. Vree (1987). The dutch parallel reduction machine project, *Proc. of Intern. Conf. on Frontiers in Computing*, Amsterdam, the Netherlands.

Barendsen, Erik and Sjaak Smetsers (1992). Graph rewriting and copying, *Technical Report 92-20*, University of Nijmegen.

Brus, T., M.C.J.D. van Eekelen, M. van Leer, M.J. Plasmeijer and H.P. Barendregt (1987). Clean - a language for functional graph rewriting, *Proc. of Conference on Functional Programming Languages and Computer Architecture (FPCA '87)*, Portland, Oregon, USA, Springer Verlag, LNCS 274, pp. 364–384.

Burstall, R.M., D.B. MacQueen and D.T. Sanella (1980). Hope: An experimental applicative language, *Proc. of The 1980 LISP Conference*, pp. 136–143.

Eekelen, M.C.J.D. van (1988). *Parallel Graph Rewriting, Some Contributions to its Theory, its Implementation and its Application*, Dissertation, University of Nijmegen.

Eekelen, M.C.J.D. van and M.J. Plasmeijer (1990). Concurrent functional programming, *Proc. of Conference on Unix & Parallelism*, pp. 75–98.

Eekelen, M.C.J.D. van, E.G.J.M.H. Nöcker, M.J. Plasmeijer and J.E.W. Smetsers (1990). Concurrent clean, version 0.6, *Technical Report 90-21*, University of Nijmegen.

Eekelen, M.C.J.D. van, M.J. Plasmeijer and J.E.W. Smetsers (1991). Parallel graph rewriting on loosely coupled machine architectures, *in:* Kaplan and Okada (eds.), *Proc. of Conditional and Typed Rewriting Systems (CTRS'90)*, Montreal, Canada, Springer Verlag, LNCS 516, pp. 354–369.

Glauert, J.R.W., J.R. Kennaway and M.R. Sleep (1987). Dactl: A computational model and compiler target language based on graph reduction, *ICL Technical Journal*.

Groningen, J.H.G. van (1990). *Implementing the abc-machine on m680x0 based architectures*, Master's thesis, University of Nijmegen.

Johnsson, Th. (1987). *Compiling Lazy Functional Programming Languages*, Dissertation, Chalmers University, Göteborg, Sweden.

Kesseler, M. (1990). Concurrent clean on transputers.

Koopman, P.W.M. and E.G.J.M.H. Nöcker (1988). Compiling functional languages to term graph rewriting systems, *Technical Report 88-1*, University of Nijmegen.

Koopman, P.W.M., M.C.J.D. van Eekelen, E.G.J.M.H. Nöcker, M.J. Plasmeijer and J.E.W. Smetsers (1990). The abc-machine: A sequential stack-based abstract machine for graph rewriting, *Technical Report 90-22*, University of Nijmegen, The Netherlands.

McBurney, D. and R. Sleep (1990). Concurrent clean on zapp, *Proc. of Second International Workshop on Implementation of Functional Languages on Parallel Architectures*, University of Nijmegen, the Netherlands, Technical Report 90-16.

Milner, R.A. (1978). Theory of type polymorphism in programming, *Journal of Computer and System Sciences*.

Mycroft, A. (1984). Polymorphic type schemes and recursive definitions, *Proc. of 6th Int. Conf. on Programming*, Eindhoven, The Netherlands, Springer Verlag, LNCS 167, pp. 217–239.

Nöcker, E.G.J.M.H. (1989). The pabc simulator, v0.5. implementation manual, *Technical Report 89-19*, University of Nijmegen.

Nöcker, E.G.J.M.H. (1990). Strictness analysis based on abstract reduction, *Proc. of Second International Workshop on Implementation of Functional Languages on Parallel Architectures*, University of Nijmegen, the Netherlands, Technical Report 90-16, pp. 297–321.

Nöcker, E.G.J.M.H. and J.E.W. Smetsers (1990). Partially strict data types, *Proc. of Second International Workshop on Implementation of Functional Languages on Parallel Architectures*, University of Nijmegen, the Netherlands, Technical Report 90-16, pp. 237–255.

Peyton Jones, S.L. and J. Salkild (1989). The spineless tagless g-machine, *Proc. of Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*, London, UK, Addison Wesley, pp. 184–201.

Smetsers, J.E.W. (1989). Compiling clean to abstract abc-machine code, *Technical Report 89-20*, University of Nijmegen.