The Concurrent Clean System

-

Functional Programming on the Macintosh

M.J. Plasmeijer,
M.C.J.D. van Eekelen, E.G.J.M.H. Nöcker, J.E.W. Smetsers,
University of Nijmegen
Department of Computer Science
Toernooiveld 1
6525 ED Nijmegen
E-mail: clean@cs.kun.nl.

FUNCTIONAL PROGRAMMING LANGUAGES

Functional programming languages are general purpose, high-level languages based on the mathematical notion of functions. A functional program consists of a set of (possibly recursive) function definitions. The execution of a program consists of the evaluation of an indicated function application.
Programs written in a functional language are generally very *compact* and also very *elegant*. This is mainly due to the availability of *pattern matching*, *guards* and *higher-order functions*. Modern functional languages use *lazy* evaluation which means that expressions are only evaluated when their values are actually needed in a calculation. This makes it possible to define *infinite data structures*. A programmer never has to worry about memory management or pointers. There is no assignment statement. As a consequence, there are no side-effects possible such that one can reason about functional programs using traditional mathematical proof techniques like induction and symbolic substitution. The expressive power is the same as with ordinary languages such as C. Another property of a functional program is that the order in which functions are evaluated cannot change the outcome of a computation. This makes functional programs also very suited for parallel execution. For all these reasons an increasing number of universities use functional languages in introductory programming courses. Functional languages are also very suited for rapid prototyping.
A disadvantage of functional languages was that programs ran very, very slow and that they consumed a large amount of memory. At several universities much attention has been paid to improve the compilation techniques which has led to good compilers for several languages (Hope (Burstall *et al.* 1980), Lml (Johnson (1984), Haskell (Hudak *et al.* (1990)). However, good compilers for personal computers such as the Macintosh were not available until now.

CONCURRENT CLEAN

The University of Nijmegen developed *Concurrent Clean*: (Brus *et al.* (1987), Nöcker *et al.* (1991)) an experimental, *higher order lazy functional programming language* suited for evaluation on a range of computer architectures varying from personal computers to parallel machine architectures.

The most important features of Concurrent Clean are:

- It is a *lazy* and *purely functional* programming language based on Term Graph Rewriting (Barendregt *et al.* (1987)).

- It is a *strongly typed* language (based on the well-known Milner (1978) type inferencing scheme) including *polymorphic types*, *abstract types*, *algebraic types* and *synonym types*, as well as basic types (*integers*, *reals*, *characters*, *booleans*, *strings, lists, tuples* and *files*).
- It has a *module structure* with *implementation modules* and *definition modules* offering a facility to implicitly and explicitly import definitions from other modules; it includes predefined modules (libraries) for basic operations (delta rules) on objects of basic types.

Example of a Concurrent Clean program: the definitions of the factorial function and the function `Map`:

```
IMPLEMENTATION MODULE example;

IMPORT delta;

RULE
::   Fac INT   ->   INT                 ;
     Fac 0     ->   1                   |
     Fac n     ->   *I n (Fac (--I n))  ;

::   Map (=> x y) [x]   ->   [y]             ;
     Map f []           ->   []              |
     Map f [a|b]        ->   [f a|Map f b]   ;

::   Start     ->   [INT]               ;
     Start     ->   Map Fac [2,3,4]     ;
```

In this program the factorial function is applied to each element of the list `[2,3,4]`. Each function definition, optionally preceded by a type specification, consists of a number of alternatives. Square brackets are used for denoting lists: `[]` is an empty list, `[a|b]` denotes a list consisting of a list `b` prefixed with an element `a`. The example also shows the use of higher order functions such as `Map`. Types of higher order functions are specified using => (prefix notation) which corresponds to -> (infix notation) as used in most other functional languages. In the example the predefined functions being used (integer multiplication (`*I`) and decrement (`--I`)) are imported with one simple import statement.

- *Annotations* can be added to a function definition. With these annotations the evaluation order can be controlled by the programmer: functions can be made (partially) *strict* instead of lazy. When a function is known to be strict in a certain argument, the argument can be evaluated before the function is called. Such functions are in general more efficient. Further speed-ups can be obtained by defining (partially) strict data structures (Nöcker & Smetsers (1990)). It is also possible to split up the execution of the program in parts that are to be evaluated *interleaved* or in *parallel*. Processes can be created dynamically with arbitrary process topologies (for instance cyclic structures). The communication between processes does not have to be explicitly defined but is handled automatically. The sending and receiving of information between processes is not explicitly defined but handled automatically.

Example to show how *divide-and-conquer* parallelism can be specified in Concurrent Clean:

```
MODULE dfib;

IMPORT delta;

RULE
::   Start -> INT     ;
     Start -> DFib 22 ;
```

```
::   Threshold -> INT ;
     Threshold -> 15  ;

::   DFib INT -> INT                                    ;
     DFib n   -> IF (<=I n Threshold)
                    (Fib n)
                    (PFib n)                            ;

::   Fib INT -> INT                                     ;
     Fib 1   -> 1                                       |
     Fib 2   -> 1                                       |
     Fib n   -> +I (Fib (-I n 1)) (Fib (-I n 2))        ;

::   PFib INT -> INT                                    ;
     PFib n   -> +I ({P} DFib (-I n 1)) (DFib (-I n 2)) ;
```

The `{P}` annotation in the definition of the function `Pfib` means that the corresponding call of `Fib` has to be evaluated in parallel. This function application is sent to another processor for evaluation. The father process will continue with the calculation of the other call of `Fib` after which it will wait until the results of parallel calculation are copied back such that addition can take place.

THE CONCURRENT CLEAN SYSTEM

The *Concurrent Clean System* is especially designed for the Macintosh following the Mac's user interface philosophy. The current version of the system (version 0.7) contains:

- A Macintosh program development environment including a project manager and a simple text-editor.
- A fast code generator for the Mac (both 68000 as 68020). Due to special compilation techniques, such as automatic strictness analysis (Nöcker (1988)), the code generator is one of the best for functional programming languages currently available (over 1.000.000 function calls per second on a Mac-fx, see Table 1).

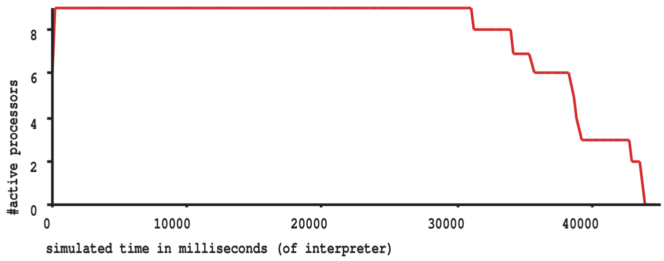
	Clean MacII-fx	Clean MacII-si	Clean Mac+	Clean Sun3	Clean (u!) Sun3	Lml Sun3	Hope Sun3	C Sun3
nfib	2.6	5.2	53	4.5	4.5	25	5.4	11
tak	2.6	5.3	53	4.9	4.9	40	7.2	11
sieve	4.4	9.4	260	8.1	6.8	25	9.1	4.5
queens	15	41	240	28	14	62	16	4.1
reverse	31	63	620	64	50	110	65	--
twice	0.86	1.8	Outofheap	1.7	0.5	SegFault	0.3	--
rnfib	6.1	13	2000	11	11	26	33	19
fastfourier	14	30	Outofheap	34	19	Outofheap	Outofheap	9

Table 1. Concurrent Clean compared with other well-known implementations using some standard benchmarks (nfib 30, tak 24 16 8, sieve 10000, queens 10, reverse 3000, twice 4 incr, rnfib 26.0 using reals, fastfourier on 8K complex numbers). The first three columns give the speed figures measured on different Macs using a 2Mb heap (only 1Mb for the Mac+). All times are in seconds. To make a proper comparison possible all programs are also tested on a SUN3/280, M68020 processor on 25Mhz clock, 2Mb heap. In Clean (u!) strictness annotations are added by the programmer. Lml is a lazy functional language, Hope is a non-lazy functional language and C is a non-lazy non-functional language.

- A linker is included that can generate stand-alone Mac applications. They can also be sub-launched from the Clean system. There is an option to generate assembly code for the Mac as well as for the Sun3.
- An interpreter is included that can simulate the execution of parallel programs; it includes facilities for tracing and debugging.
- The interpreter can produce several kinds of statistical information about the simulated behaviour of parallel execution. With an additional application especially designed for the Mac this information can be graphically displayed and the run-time behaviour can be examined.

The `Pfib` example above executed by the simulator simulating a parallel machine with 9 processors produces the following results:

```
Speedup          776%
#processes       34
#suspensions     33
communication    2244 bytes
```



AVAILABILITY

Version 0.7 of the Concurrent Clean System with all the features mentioned above is now available for both the Macintosh as well as Sun3 and is distributed *free* for educational and research purposes. It can be obtained via anonymous FTP (phoibos.cs.kun.nl (131.174.81.1) in the directory pub/Clean) or by electronic mail (clean@cs.kun.nl) or by sending two floppy disks (for the Mac) or a cartridge (for the SUN3) to the address above. The Macintosh version runs on any Mac with system 6.0 (or higher), needs at least 1.5 megabyte and requires a multi-finder to make sub-launching possible.

FUTURE PLANS

For the final version of the Concurrent Clean System the following extensions are planned:

- the efficiency of the sequential code will be further improved;
- more syntactical sugar will be added (guards, infix notation, overloading);
- more sophisticated annotations for parallel evaluation will be added;
- an even more liberal type system based on intersection types (Coppo (1980)) will be added;
- IO facilities that make menu and dialogue handling possible in Clean applications;
- the possibility to import Miranda™ scripts (Turner (1985));
- a parallel version for the Mac, using several Macs connected via the Appletalk network;
- a code generator for our 64-node Parsytec Transputer system.

CONCLUSIONS

With the Concurrent Clean System (parallel) functional programming has become available on the Macintosh. The system is used for teaching students functional programming in general as well as for studying the behaviour of functional programs on parallel machine architectures.

Miranda™ is a trademark of Research Software Ltd.

Combined with a student textbook version 1.0 of the Concurrent Clean System will be made widely available in the beginning of 1992 (Plasmeijer & van Eekelen (1990)).

REFERENCES

- Barendregt H.P., Eekelen M.C.J.D. van, Glauert J.R.W., Kennaway J.R., Plasmeijer M.J., Sleep M.R. (1987), 'Term Graph Rewriting, Proceedings of Parallel Architectures and Languages Europe (PARLE), part II, Eindhoven, The Netherlands, LNCS Vol. 259, pp. 141-158, June 1987.
- Brus, T., Eekelen, M.C.J.D. van, Leer, M. van, Plasmeijer, M.J. (1987). Clean - A Language for Functional Graph Rewriting. Proc. of the Third International Conference on Functional Programming Languages and Computer Architecture (FPCA '87), Portland, Oregon, USA, *Springer Lec. Notes on Comp.Sci.* 274, 364 - 384.
- Burstall R.M., MacQueen D.B., and Sanella D.T. (1980), 'Hope: An Experimental Applicative Language', Proceedings of the 1980 LISP Conference, 136 - 143.
- Coppo, M., Dezani-Ciancaglini, M. An Extension of the Basic Functionality Theory for the λ-calculus. *Notre Dame, Journal of Formal Logic 21 (4)*, 1980, page 685-693.
- Hudak, P., Wadler, Ph., Arvind, Boutel, B., Fairbairn, J., Fasel, J., Hammond, K., Hughes, J., Johnsson, Th., Kieburtz, D., Nikhil, R., Peyton Jones, S., Reeve, M., Wise, D., Young, J. (1990). Report on the programming language Haskell. Version 1.0.
- Johnsson Th. (1984). Efficient compilation of lazy evaluation. Proceedings of the ACM SIGPLAN '84, Symposium on Compiler Construction. *SIGPLAN Notices* 19/6.
- Milner R.A. (1978). Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, Vol. 17, no. 3, 348 - 375.
- Nöcker E.G.J.M.H. (1988). 'Strictness Analysis based on Abstract Reduction of Term Graph Rewrite Systems. in Proceedings of the Workshop on Implementation of Lazy Functional Languages', University of Göteborg and Chalmers University of Technology, Programming Methodology Group, Report 53.
- Nöcker E.G.J.M.H., Smetsers J.E.W., (1990). 'Partially Strict Data Types', in Proceedings of the Second International Workshop on Implementation of Functional Languages on Parallel Architectures, pp. 237-255, Technical Report no. 90-16, October 1990, University of Nijmegen.
- Nöcker E.G.J.M.H., Smetsers J.E.W., Eekelen M.C.J.D. van, Plasmeijer M.J. (1991). 'Concurrent Clean', to be published in the Proceedings of the Conference on Parallel Architectures and Languages Europe 1991 (PARLE).
- Plasmeijer M.J., Eekelen M.C.J.D. van (1990). *Functional Programming and Parallel Graph Rewriting*. Draft Lecture Notes. University of Nijmegen. Final version to be published by Addison Wesley, spring 1992.
- Turner, D.A. (1985) Miranda: A non-strict functional language with polymorphic types. Proc. of the conference on Functional Programming Languages and Computer Architecture, *Springer Lec. Notes Comp. Sci.* 201, 1 - 16.