URL: http://www.elsevier.nl/locate/entcs/volume2.html 7 pages

Clean: a Programming Environment Based on Term Graph Rewriting

M.J. Plasmeijer

Faculty of Mathematics and Informatics
Katholieke Universiteit Nijmegen
Toernooiveld 1 6525 ED, Nijmegen, The Netherlands

Abstract

The main features of the lazy functional language Concurrent Clean and of its semantics based on Term Graph Rewriting are presented.

1 History

The pure, lazy functional language Concurrent Clean (version 1.0) is a major upgrade of the previous release (0.84b) of Clean ([3,10,11]). Clean was originally designed as an experimental intermediate language and deliberately kept syntactically as poor as possible to be able to focus on the essential language and implementation issues. This strategy enabled us to study new concepts (such as term graph rewriting [1], lazy copying [4], abstract reduction [8], uniqueness typing [2]) without too much implementation effort. The ideas were tested in the Clean compiler which could be used on small machines and produced state-of-the-art code [13]. The consequence was that people started to use Clean to construct large applications even though Clean was actually not intended as a programming language. So, it became necessary to turn the experimental intermediate language into a proper practical applicable general purpose functional programming language suited for the development of real world applications.

1.1 What is special about Concurrent Clean 1.0?

Compared with the previous version of Clean a lot of new features are added based on our own experience with writing complex applications (such as the new Clean I/O system). Many of the added language constructs are similar to those commonly found in other modern lazy functional languages (such as Miranda [15], SML [5], Haskell [7]. People familiar with these languages will have no difficulty to program in Clean and we hope that they enjoy the compilation speed and quality of the produced code. In addition Clean offers a couple of very special features. Of particular importance for practical use is Cleans' uniqueness typing enabling the incorporation of destructive updates of arbitrary objects within a pure functional framework and the creation of direct interfaces with the outside world. Cleans' "unique" features have made it possible to predefine (in Clean) a sophisticated and efficient I/O library. The Clean I/O library enables a Clean programmer to specify interactive window based I/O applications on a very high level of abstraction. The library forms a platform independent interface to window systems which makes it possible to port window based I/O applications written in Clean without modification of source code. In Clean it is possible to create processes. The new Clean I/O library takes advantage of this feature such that it is now also has become possible to develop distributed executing interactive applications running on several PC's/workstations connected in a network. The applications can communicate via asynchronous as well as synchronous message passing. Such a distributed application can be developed on one processor on which the processes will run in an interleaved fashion. This is very handy for testing. The new Clean compiler still combines fast compilation with the generation of efficient code and is available on an increasing number of platforms (Mac, PC, Sun).

1.2 How to obtain Clean

Concurrent Clean and the Conurrent Clean Program Development system be used free of charge for educational purposes only. They can be obtained

- via World Wide Web (www.cs.kun.nl/~clean) or
- via ftp (ftp.cs.kun.nl in directory pub/Clean).

It is allowed to copy the system again for educational purposes only under the condition that the whole distribution for a certain platform is copied, including help files, the language report and the copyright notices. For any commercial use of Clean a commercial license is required, which is not free of charge. Information about commercial licenses can be obtained by contacting Rinus Plasmeijer (rinus@cs.kun.nl). For commercial users additional utility software is supplied and full technical support is given to assist to incorporate Clean and Clean applications in a specific environment.

1.3 Key design rules used for Clean 1.0

- The language must be purely functional, higher order and lazy;
- The semantics of the language must be based on graph rewriting systems;
- The language must be suitable for writing real world applications in a very compact and readable style;
- It must be possible to create programs with an efficiency comparable with C;
- Direct and efficient interfacing with the non-functional world must be possible;
- One must be able to control the time and space efficiency of the program;
- Parallel and distributed evaluation of programs must be possible;

- Program components must be re-usable;
- A program (including window based interactive programs) must be fully portable.

1.4 Short summary of the features of Clean 1.0

The most important features of Clean are:

- Clean is a lazy, pure, higher order functional programming language with explicit graph rewriting semantics; one can explicitly define the sharing of structures (cyclic structures as well) in the language;
- Although Clean is by default a lazy language one can smoothly turn it into a strict language to obtain optimal time/space behaviour: functions can be defined lazy as well as (partially) strict in their arguments; any (recursive) data structure can be defined lazy as well as (partially) strict in any of its arguments;
- Clean is a strongly typed language based on an extension of the well-known Milner/Hindley type inferencing scheme [9,6] including the common polymorphic types, abstract types, algebraic types, and synonym types extended with a restricted facility for existentially quantified types;
- Type classes and type constructor classes are provided to make overloaded use of functions and operators possible.
- Clean offers the following predefined types: integers, reals, booleans, characters, strings, lists, tuples, records, arrays and files;
- Cleans' key feature is a polymorphic uniqueness type inferencing system, a special extension of the Milner/Hindley type inferencing system allowing a refined control over the single threaded use of objects; with this uniqueness type system one can influence the time and space behaviour of programs; it can be used to incorporate destructive updates of objects within a pure functional framework, it allows destructive transformation of state information, it enables efficient interfacing to the non-functional world (to C but also to I/O systems like X-Windows) offering direct access to file systems and operating systems;
- Clean is a modular language allowing separate compilation of modules; one defines implementation modules and definition modules; there is a facility to implicitly and explicitly import definitions from other modules;
- Clean offers a sophisticated I/O library with which window based interactive applications (and the handling of menus, dialogues, windows, mouse, keyboard, timers and events raised by sub-applications) can be specified compactly and elegantly on a very high level of abstraction;
- Specifications of window based interactive applications can be combined such that one can create several applications (sub-applications or light-weight processes) inside one Clean application. Automatic switching between these sub-applications is handled in a similar way as under a multi-finder (all low level event handling for updating windows and switching

between menus is done automatically); sub-applications can exchange information with each other (via files, via clipboard copy-paste like actions using shared state components, via asynchronous message passing) but also with other independently programmed (Clean or other) applications running on the same or even on a different host system;

- Subapplications can be created on other machines which means that one can define distributed window based interactive Clean applications communicating e.g. via (a)synchronous message passing and remote procedure calls across a local area network;
- Dynamic process creation is possible; processes can run interleaved or in parallel; arbitrary process topologies (for instance cyclic structures) can be defined; the interprocess communication is synchronous and is handled automatically simply when one function demands the evaluation of its arguments being calculated by another process possibly executing on another processor;
- Due to the strong typing of Clean and the obligation to initialize all objects being created run-time errors can only occur in a very limited number of cases: when partial functions are called with arguments out of their domain (e.g. dividing by zero), when arrays are accessed with indices out-of-range and when not enough memory (either heap or stack space) is assigned to a Clean application;
- Clean 1.0 is supported on several platforms.

2 Term Graph Rewrite Semantics

The semantics of Clean is based on Term Graph Rewriting [1,12]. This means that functions in a Clean program semantically work on graphs instead of the usual terms. In many cases the programmer does not need to be aware of the fact that he/she is manipulating graphs. Evaluation of a Clean program takes place in the same way as in other lazy functional languages.

The main "difference" between Clean and other languages is that when a variable occurs more than once in a function body, the semantics prescribe that the actual argument is shared (the semantics of most other languages do not prescribe this although it is common practice in any implementation of a functional lan-guage). Furthermore, one can label any expression to make the definition of cyclic structures possible. So, people familiar with other functional languages will have no problems writing Clean programs.

When larger applications are being written, or, when Clean is interfaced with the non-functional world, or, when efficiency counts, or, when one simply wants to have a good understanding of the language it is good to have some knowledge of the basic semantics of Clean which is based on term graph rewriting. An extensive treatment of the underlying semantics and the implementation techniques of Clean can be found in [11].

2.1 Term graph rewriting and Clean

A Clean program basically consists of a number of graph rewrite rules (function definitions) which specify how a given graph (the initial expression) has to be rewritten.

A graph is a set of nodes. Each node has a defining node-identifier (the node-id). A node consists of a symbol and a (possibly empty) sequence of applied node-id's (the arguments of the symbol). Applied node-id's can be seen as references (s) to nodes in the graph, as such they have a direction: from the node in which the node-id is applied to the node of which the node-id is the defining identifier.

Each graph rewrite rule consists of a left-hand side (the pattern) and a right-hand side (rhs) consisting of a (the contractum) or just a single node-id (a redirection). In Clean rewrite rules are not comparing: the left-hand side (lhs) graph of a rule is a tree, i.e. each node identifier is applied only once, so there exists exactly one path from the root to a node of this graph.

A rewrite rule defines a (partial) function. The function symbol is the root symbol of the left-hand side graph of the rule alternatives. All other symbols that appear in rewrite rules, are constructor symbols.

The program graph is the graph that is rewritten according to the rules. Initially, this program graph is fixed: it consists of a single node containing the symbol Start, so there is no need to specify this graph in the program explicitly. The part of the graph that matches the pattern of a certain rewrite rule is called a redex (reducible expression). A rewrite of a redex to its reduct can take place according to the right-hand side of the corresponding rewrite rule. If the right-hand side is a contractum then the rewrite consists of building this contractum and doing a redirection of the root of the redex to root of the right-hand side. Otherwise, only a redirection of the root of the redex to the single node-id specified on the right-hand side is performed. A redirection of a node-id n1 to a node-id n2 means that all applied occurrences of n1 are replaced by occurrences of n2 (which is in reality commonly implemented by overwriting n1 with n2).

A reduction strategy is a function that makes choices out of the available redexes. A reducer is a process that reduces redexes that are indicated by the strategy. The result of a reducer is reached as soon as the reduction strategy does not indicate redexes any more. A graph is in normal form if none of the patterns in the rules match any part of the graph. A graph is said to be in root normal form when the root of a graph is not the root of a redex and can never become the root of a redex. In general it is undecidable whether a graph is in root normal form.

A pattern partially matches a graph if firstly the symbol of the root of the pattern equals the symbol of the root of the graph and secondly in positions where symbols in the pattern are not syntactically equal to symbols in the graph, the corresponding sub-graph is a redex or the sub-graph itself is partially matching a rule. A graph is in strong root normal form if the graph does not partially match any rule. It is decidable whether or not a graph is in strong root normal form. A graph in strong root normal form does not partially match any rule, so it is also in root normal form.

The default reduction strategy used in Clean is the functional reduction strategy. Reducing graphs according to this strategy resembles very much the way execution proceeds in other lazy functional languages: in the standard lambda calculus semantics the functional strategy corresponds to normal order reduction. On graph rewrite rules the functional strategy proceeds as follows: 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 to strong root normal form of arguments is forced when an actual argument is matched against a corresponding non-variable part of the pattern. A formal definition of this strategy can be found in [14].

References

[1] Barendregt H.P., Eekelen M.C.J.D. van, Glauert J.R.W., Kennaway J.R., Plasmeijer M.J. and Sleep M.R. (1987). *Term graph rewriting*, In *Proc. Parallel Architectures and Languages Europe* (PARLE '87), Eindhoven (Bakker J.W. de, Nijman A.J. and Treleaven P.C., eds.), LNCS, 259, II, pp. 141–158. Berlin: Springer-Verlag.

[2] Barendsen E. and Smetsers J.E.W. (1993). *Conventional and uniqueness typing in graph rewrite systems* (extended abstract), in: R.K. Shyamasundar (ed.) Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science, Bombay, India, LNCS 761, Springer-Verlag, pp 41–51.

[3] Brus T., Eekelen M.C.J.D. van, Leer M. van, Plasmeijer M.J. and Barendregt H.P. (1987). *Clean - a language for functional graph rewriting*. In Proc. Conference on Functional Programming Languages and Computer Architecture (FPCA '87), Portland OR (Kahn G., ed.), LNCS, 274, pp. 364–384. Berlin: Springer-Verlag.

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

[5] Harper R., MacQueen D. and Milner R. (1986). *Standard ML*. Internal Report ECS-LFCS-86-2, Edinburgh University.

[6] Hindley J.R. (1969). *The principal type scheme of an object in combinatory logic*. Trans American Mathematical Society, 146, pp. 29–60.

[7] Hudak P., Peyton Jones S., Wadler Ph., Boutel B., Fairbairn J., Fasel J., Hammond K., Hughes J., Johnsson Th., Kieburtz D., Nikhil R., Partain W. and Peterson J. (1992). *Report on the programming language Haskell*. ACM SigPlan Notices, 27, (5), pp. 1–164.

[8] Nöcker E.G.J.M.H. (1993). *Strictness analysis using abstract reduction*. In Proc. Conference on Functional Programming Languages and Computer Architectures (FPCA '93), Copenhagen. ACM Press.

[9] Milner R.A. (1978). *Theory of type polymorphism in programming*. J of Computer and System Sciences, 17, (3), pp. 348–375.

[10] Nöcker E.G.J.M.H., Smetsers J.E.W., Eekelen M.C.J.D. van and Plasmeijer M.J. (1991). *Concurrent Clean*. In Proc. Parallel Architectures and Languages Europe (PARLE '91), Eindhoven (Aarts E.H.L., Leeuwen J. and Rem M., eds.), LNCS, 506, pp. 202–219. Berlin: Springer-Verlag.

[11] Plasmeijer M.J. and Eekelen M.C.J.D. van (1993). *Functional Programming and Parallel Graph Rewriting*. Addison Wesley.

[12] Sleep M.R., Plasmeijer M.J. and Eekelen M.C.J.D. van, eds. (1993). *Term Graph Rewriting*. New York: John Wiley.

[13] Smetsers J.E.W., Nöcker E.G.J.M.H., Groningen J.H.G. van and Plasmeijer M.J. (1991). *Generating efficient code for lazy functional languages*. In Proc. Conference on Functional Programming Languages and Computer Architecture (FPCA '91), Cambridge MA (Hughes J., ed.), LNCS, 523, pp. 592–617. Berlin: Springer-Verlag.

[14] Toyama Y., Smetsers J.E.W., Eekelen M.C.J.D. van and Plasmeijer M.J. (1993). *The functional strategy and transitive term rewriting systems*. In *Term Graph Rewriting* (Sleep M.R., Plasmeijer M.J. and Eekelen M.C.J.D. van, eds.). New York: John Wiley.

[15] Turner D.A. (1985). *Miranda: a non-strict functional language with polymorphic types*. In Proc. Conference on Functional Programming Languages and Computer Architecture, Nancy, France (Jouannaud J.P., ed.), LNCS, 201, pp. 1–16. Berlin: Springer-Verlag.