Term Graph Rewriting and Mobile Expressions in Functional Languages

Rinus Plasmeijer and Marko van Eekelen

Computing Science Institute
University of Nijmegen, the Netherlands
{rinus, marko}@cs.kun.nl

Abstract. CLEAN is a functional language based on Term Graph Rewriting. It is specially designed to make the development of real world applications possible by using a pure functional language.
In this paper we first give a short overview of the most important basic features of the language CLEAN among which it's Term Graph Rewriting semantics. Of particular importance for practical use is CLEAN's uniqueness typing enabling destructive updates of arbitrary objects and the creation of direct interfaces with the outside world, all within a purely functional framework.
After this overview we will focus on a new language feature, which is currently being added. The new version of CLEAN offers a hybrid type system with both static as well as dynamic typing. Expressions, which are dynamically typed, are called Dynamics. With help of Dynamics one can create mobile expressions, which can be passed to other CLEAN applications. Dynamics can be used to make plug-ins which will be type checked at run-time. Typically, 30% of the code of an application is needed for storing (converting data to string) and retrieving (by means of a parser) of data. With Dynamics one can store and retrieve not only data but also code (!) with just *one* instruction.
The implementation effort needed to support Dynamics is quite large: it not only involves dynamic type checking but also dynamic type unification, dynamic linking, just-in-time compilation, coding techniques for data and version management of code segments.

1 Clean: a functional language for real world applications

The CLEAN [23] system includes a very fast compiler (typically it compiles one to two orders of magnitude faster than comparable compilers for pure and lazy functional languages) and it generates state-of-the-art, native code. The system comprises an Integrated Development Environment, an editor, a project manager, a linker, a time and space profiling tool and a tool for creating interfaces to C. Almost all of this software is written in the language CLEAN itself. CLEAN is available on a large variety of platforms such as WINDOWS '95 / '98 / '2000 / NT, MACOS, UNIX (SUN) and LINUX (PC). The CLEAN software can be downloaded from the net (*www.cs.kun.nl/~clean*).

Clean is currently mainly used in R&D environments. We have over 1000 administrated users (about 1/3 from industry). There is a Clean mailing list via which users frequently communicate about the use of Clean. Clean is commercially exploited by the Nijmegen University spin-off company Hilt. Among the commercial users of Clean are the Canadian Telecom Company Newbridge, Microsoft, the Dutch companies Philips, ABN-AMRO, Hollandse Signaal and the Dutch Department of Public Works.

Applications written in Clean include a software development monitoring tool, a Java Applet generator, a music composition program, a machine code linker, a platform independent I/O library, an editor, an ordered linear resolution prover, an integrated software development environment, a projective geometry demonstration program, an audio-set software generation and configuration tool, a sentence generator for spoken text, a network process profiling tool, a traffic light simulation tool and a game library for the development of 2D platform games [26].

Since the Clean I/O library is available on a large variety of platforms, it provides a platform independent interface for reactive Clean programs. Interactive window based Clean programs can be ported to any of these platforms *without any modification of source code*. Programs retain the specific "look and feel" offered by the platform being used.

Haskell [13] and Clean [9] [20] [22] developed independently as descendants of the language Miranda [24] and influenced by the language Gopher (type classes for overloading [16]). The first publication on Clean dates back to 1987 [9] at the Functional Programming and Computer Architecture conference in Oregon where the Haskell committee was formed.

Clean is a state-of-the-art lazy and pure functional language and as such it offers features like higher order functions, currying, lazy evaluation, (cyclic) sharing, lambda expressions and local definitions (where and let), guards and case expressions, patterns, list and array comprehensions, strong typing with Milner/Mycroft type inference (with polymorphic types, abstract types, algebraic types, synonym types) extended with existentially quantified types, overloading via type classes and type constructor classes, predefined types and type constructors (integers, reals, Booleans, characters, files, lists, tuples, records, arrays), strictness annotations in (function and data) type definitions, separate compilation of modules (with implementation and definition modules with implicit or explicit imports). Apart from small differences the most important differences between Clean and Haskell are that Clean has graph rewriting semantics, offers unique typing, and has a sophisticated library for defining window based interactions. These distinctive features of Clean are explained in more detail below.

2 The importance of graph rewriting

There are several models of computation that can be viewed as a theoretical basis for functional programming. The *lambda calculus* (see [6]), being a well-understood mathematical theory, is traditionally considered to be the purest foundation for modern functional languages like SCHEME [11], ML [10], HASKELL [13], MIRANDA [24] and CLEAN [9] [20] [22]. And indeed, the merits of many programming languages concepts have successfully been investigated in the (typed) lambda calculus. However, *all* state-of-the-art *implementations* of functional languages are *not* based on lambda calculus but on graph rewriting. Consider with λ-calculus semantics the following definition: `square x = x * x`. Then, reducing the following function application `square (1 + 1)` would give `(1 + 1) * (1 + 1)` with multiple copies of the argument expression. In practice such an argument may contain a huge calculation, which is copied many times. Graph rewriting semantics avoid this multiple evaluation by *sharing* the argument in the graph structure. Since the argument is then addressed via two pointers, the evaluation takes place only once. The resulting evaluation order corresponds to call-by-need semantics.

Clearly, programmers writing real world applications will have to address much more practical aspects that are all related to graphs and graph rewriting: they worry about data structures, sharing, cycles, space consumption, efficiency, updates, input/output, interfacing with C, and so on. Many of these problems and solutions can be understood better when the semantics of the functional language are extended incorporating graph structures. This is the reason we have sought for a model of computation that is sufficiently elegant and abstract, but at the same time incorporates mechanisms that are more realistic with respect to actual implementation techniques. *Graph rewriting systems* [7] extend λ-calculus with the explicit notions of pattern matching and sharing.

Graph rewriting has been proven to serve very well as a uniform framework: the *programmer* uses it to reason about the program and to control time and space efficiency of functions and graph structures; the *implementer* of the compiler uses it to design optimisations and analysis techniques that are correct with respect to the graph rewriting theory; the *theoretician* uses it to build theory for complex analysis of graph rewriting systems used in concrete implementations; the *language designer* uses it to base the functional language constructs directly on the graph rewriting semantics.

In our opinion it was the availability of this common framework in CLEAN that played the key role in various activities that usually are far apart: extending the language with new vital constructs that otherwise would never have been found, keeping the compiler fast and correct, enabling the programmer to write efficient programs and keeping the theory to the point.

The programmer's choice of representation, e.g. the explicit use of a cycle, can drastically influence the algorithmic complexity (e.g. bringing it down from exponential to polynomial). The use of graph rewriting semantics makes it possible for the programmer to model this choice.

3 The importance of purity

An important aspect of CLEAN is that it is a *pure* functional language, like MIRANDA and HASKELL. Examples of impure functional languages are LISP [18], SCHEME [11], ML [10], CAML [17] and ERLANG [5]. In a pure language the result of a function application will, under *all* conditions, be determined solely by the value of its arguments. This key property is called *referential transparency*. A consequence is that it never makes any difference where and under what conditions a function is applied. A function will *always* react in exactly the same way. This has important consequences. Assignments do not exist and side effects cannot occur. A functional programmer can look at a piece of code and simplify or change it without the need to carefully scan every other function or procedure for context dependency. Reasoning about the result of a function can be done by *uniformly substituting* its definition at the place of the application regardless of the context. A C or JAVA programmer will always risk the surprise that calling the same function twice has totally different effects. In CLEAN, parallel and distributed evaluation of *any* function application is allowed without worrying about changing the outcome of a program. In mathematics referential transparency and uniform substitution are two of the most fundamental properties of reasoning which are used in all fields of mathematics (and which have been used extensively by every high school student).

One might wonder how on earth one can write serious applications in a functional language and retain purity? For instance, one certainly would like to be able to change the contents of a file in a destructive way. Let's look at a simple example: assume that one would allow an impure function `fwritec`, which as a side effect appends a character to a file on disk returning the modified file. For example, the function `fwritec` can be used to construct the following function `AppendAandB` which returns a tuple:

```
AppendAandB file
  = (fwritec 'a' file, fwritec 'b' file)  // illegal in Clean !
```

The contents of the resulting tuple will depend on the order in which the two applications of `fwritec` will be evaluated. If `fwritec 'a'` file is evaluated before `fwritec 'b' file` then `'a'` is appended to the file before `'b'`. If the function applications are evaluated the other way around `'b'` will be written before `'a'`. This violates the rule of referential transparency: the result of a function application is not solely determined by the value of the arguments but also depends on the context, in this case the order in which the functions are evaluated. The purity is lost and the example illustrates that it is indeed hard to reason about the effect of such functions.

In the first generation of functional languages one simply did not know how to solve this problem and gave up purity. With this many nice mathematical properties are lost as well: mathematical analysis and every day reasoning about a program becomes almost as hard as in imperative languages such as C. One has to establish for each function whether it has such an impure aspect (or calls a function which has) before one can continue with standard reasoning.

If one is not prepared to give up purity then definitions such as AppendAandB have to be made impossible. Observe that the problem is caused by having several dangerous function applications (fwritec), which can be applied at the same time on the same object (file).

The solution taken in CLEAN is based upon the observation that the problem does not occur when there is only one pointer to an updateable object (such as a file). The previous example is disallowed because it makes two copies of the file pointer. A CLEAN programmer can explicitly pass around destructively updateable objects like any other object but he can only do this in a safe manner such that no violation of referential transparency is possible. This is guaranteed by the uniqueness type system of CLEAN.

In HASKELL the problem is solved in the following way. A program yields a higher order function (e.g. fwritec 'a') and the system applies this function to the hidden state (via a so-called *monad*) to be updated (e.g. file). By using function composition a whole sequence of state transitions can be requested. This system is simple but has the disadvantage that all objects to be destructively updated must be maintained by the system in a single state, which is kept hidden for the programmer. Clean does not have this restriction. One can have arbitrary states, which can be passed around explicitly. Such a state can be fractured into independent parts (e.g. distinct variables for the file system and the event queue). For a comparison on the two approaches see [25].

4 Uniqueness typing

In order to determine (and specify) that an argument is passed in such a way that the required updates are possible and referential transparency is maintained, a type system has been added to CLEAN which derives so-called *uniqueness properties* [8]. A function is said to have an argument of *unique type* if there will be just a single reference to the argument upon evaluation of the function. Uniqueness typing can be seen as linear logic [14] extended with sub typing and with a strategy-aware reference analysis. It is important to realize that talking about references to nodes is very natural in CLEAN since CLEAN is based on term graph rewriting. From its semantics it directly follows that it is safe for the function to re-use the memory consumed by the argument to construct the function result. For file I/O this means that the function fwritec should demand its second argument to be of unique type. In the type this uniqueness is expressed with an * attached to the conventional type. Consequence: the result of applying fwritec (the new file) can be constructed by destructively updating the unique argument (the old file) as one is used to.

The function AppendAandB defined previously will not type check. But we can easily write a function that performs the two updates in sequence.

```
fwritec:: Char *File -> *File
// type of predefined function which appends a character to a file
```

```
AppendAbeforeB:: *File -> *File
AppendAbeforeB file = fileAB
where
      fileA  = fwritec 'a' file    // append 'a' to the file
      fileAB = fwritec 'b' fileA   // then append 'b' to the file
```

To be able to write these definitions in a more natural order, CLEAN offers a special let expression, indicated by a `#`. It's scope rules allows to reuse names. The function `AppendAbeforeB` can also be defined as follows:

```
AppendAbeforeB:: *File -> *File
AppendAbeforeB file
#     file = fwritec 'a' file   // append 'a' to the file
      file = fwritec 'b' file   // then append 'b' to the file
=     file                      // and return the resulting file
```

The uniqueness type system is an extension on top of the conventional type system. The uniqueness type attributes can be inferred or checked by the compiler, as all other type information. Offering a non-unique object to a function that requires a unique one is not type correct. Offering a unique argument if a function requires a non-unique one is fine: the type system can coerce a unique object to a non-unique one.

4.1 Using Uniqueness Information

The CLEAN uniqueness type system is very powerful and flexible, it can be used to solve several problems.

First of all it can be used for *interfacing* the pure functional world in an efficient way with the impure world outside. For any (impure) foreign function, method or procedure an interface function can be written in CLEAN. The object updated destructively in the imperative world can be protected by a uniqueness type in the functional world ensuring that within CLEAN the object can only single-threadedly be passed around from function to function retaining the purity of the language. In this way external objects which have an inherently unique physical representation (such as a file) can be rewritten, a record in a database can be updated, a picture in a window on a screen can be animated. For the language C a tool is available to easily generate interface functions.

Uniqueness typing can also be used to make a functional program more efficient allowing *reusing memory* of predefined and user defined data structures. This can give a huge gain in efficiency. For instance, CLEAN offers the predefined type array and functions with which unique arrays can be destructively manipulated (as efficient as in C, about 20 times faster than arrays in the Glasgow HASKELL compiler [15]).

CLEAN's "unique" features have made it possible to predefine (in CLEAN) a sophisticated and efficient I/O library giving a program access to a unique outside world and its unique sub-components (e.g. file system, event queue, operating system). The I/O library [2] [3] [4] written in CLEAN enables a CLEAN programmer to *specify interactive window based I/O applications* on a very high level of abstraction.

5 The importance of mobile expressions

With uniqueness typing, Input/Output can be incorporated in a pure functional language without any problem. Once you have got the technical ability to perform I/O, you want to use the power of functional languages to do it more elegantly and easily as one is traditionally used to.

In the average application, 30% of the program code is used for doing trivial I/O. When data is written to a file it has to be converted to an ASCII string. When data is read one needs a parser to check the input string. Although functional languages offer powerful language features to make life easier (type classes for overloading of I/O primitives, parser combinators for easy construction of parse functions), still a lot of code has to be written. Would it not be nice to read and write complicated data structures from and to a file with just one instruction? In the functional world the difference between data and code is much smaller than in the traditional imperative world: functions are first class citizens. So, why not have the possibility to communicate any expression (containing data as well as code) with just one instruction? And why not communicate with the same ease an arbitrary expression from one (distributed executing) CLEAN application to another? Such *mobile expressions* can be used to realize plug-ins which can be dynamically added to a running application.

5.1 Cleans' Hybrid Type System

When distributed CLEAN applications communicate expressions with each other, it is of course necessary to guarantee the type safety of the messages. Distributed applications are generally not developed at the same moment such that the consistency of the messages communicated between them cannot be checked statically by inspecting the source code. So, we need a dynamic type system to check the type consistency of the mobile expressions at run-time. But we do not want the entire CLEAN language to become a dynamically typed system as well. Static type checking has too many advantages, which we want to retain: error reporting in a stage as early as possible, program code is better readable, much more efficient code can be generated. To get the best of both worlds we need a *hybrid type system* in which some parts of a program are type checked dynamically, while the largest part is still checked statically.

The concept of *objects with dynamic types*, or *dynamics* for short, as introduced by Abadi et al. [1], can provide such an interface ([21]). We distinguish between expressions on which only static type checks are performed (*statics*) and expressions on which dynamic type checks are performed (*dynamics*). Values of this dynamic type are, roughly speaking, pairs of a value and an encoding of its corresponding static type, such that both the value as well as its type can be inspected at run-time.

From a statically point of view all dynamics belong to one and the same static type: the the type Dynamic. Applications involving dynamics are no longer checked statically, but type checks are deferred until run-time. Almost any CLEAN expression can be explicitly projected from the statical into the dynamical world and vice versa.

5.2 Converting Statics into Dynamics

The projection from the statical to the dynamical world is done by pairing an expression with an encoding of its type. The expression is *wrapped* into a container. The type of the expression is hidden from the statical world and the container receives the statical type Dynamic.

A static expression of type T can be changed into a dynamic expression of static type Dynamic using the keyword dynamic. In principle, an expression of any type can be turned into a dynamic, e.g. functions of polymorphic type and functions working on unique types. Examples of expressions of type Dynamic are:

```
(dynamic True    :: Bool)       :: Dynamic
(dynamic fib     :: Int -> Int) :: Dynamic
(dynamic reverse :: [a] -> [a]) :: Dynamic
```

For instance, in the example above, the function reverse of static polymorphic type [a]->[a] is converted into a dynamic using the keyword dynamic. The resulting expression (the container containing the pair reverse and the encodng for its type [a]->[a] to be used for type checking at run-time) is of static type Dynamic.

Notice that CLEAN has a type inferrencing system, so none of the static types need to be specified explicitly. Static types can be left out and will automatically be inferred by the static type system. So, instead of the expressions above, one can also just write down:

```
dynamic True
dynamic fib
dynamic reverse
```

5.3 Converting Dynamics into Statics

Next to projection into the dynamic world, there has to be a mechanism to inspect expressions of type Dynamic and retrieve their original encoded static type such that this type information can be used in the statically typed world again. For this purpose the pattern match mechanism of CLEAN has been extended to describe pattern matching on *types* as well. An example of such a dynamic pattern match on types is:

```
SnapShot :: Dynamic -> JPeg
SnapShot (pict ::JPeg) = pict
SnapShot (movie::MPeg) = toJPeg movie
SnapShot  else         = DefaultJPegPicture
```

If the type encoded in the dynamic matches the statically specified type in the pattern, and all other patterns match as well, the corresponding rule alternative is chosen. The type demanded in a pattern on the left-hand-side can now safely be assumed in the right-hand-side of the corresponding rule alternative. The type correctness of this can all be checked statically, since the type pattern is explicitly specified in the pattern and therefore known statically.

Type Pattern Variables The type patterns need not fully specify the demanded type: they may include *type pattern variables*, which match any sub expression of the dynamic's type. If such a match has been successful, the sub expressions are bound to the type pattern variables they have matched. So, a full-blown run-time unification is used during matching of dynamics. A successful unification leads to a substitution for the type pattern variables and possibly for the (polymorphic) variables of the actual type.

The following function is polymorphic in the types of its arguments (and its result). It checks whether its first argument is a function and whether the type of its second argument matches the input type of that function:

```
dynamicApply :: Dynamic Dynamic -> Dynamic
dynamicApply (f::a->b) (x::a) = dynamic (f x::b)
dynamicApply else             = dynamic "dynamic type error"
```

Now

```
Start = dynamicApply (dynamic fib::Int->Int) (dynamic 7::Int)
```

will reduce to

```
dynamic 21::Int
```

and

```
Start =
dynamicApply (dynamic reverse::[a]->[a]) (dynamic [1,2,3]::[Int])
```

will reduce to

```
dynamic [3,2,1]::[Int]
```

In Pil [21] it is shown that the type patterns to match on can be generalized (by using a special kind of overloading) such that type restriction imposed on a dynamic type is determined by the static context in which the function is used (so called Type Dependent Functions). Type Dependent Functions allow us to bring the types of dynamics locally specified into the scope of the type of the corresponding function.

```
lookup :: [Dynamic] -> a | TC a
lookup [(x::a):xs] = x
lookup [x:xs]      = lookup xs
lookup [ ]         = abort "dynamic type error in lookup function"
```

This more powerful form of abstraction can be very convenient. For instance, the lookup function above will lookup the first element in the list of dynamics that is of the required type. This type will depend on the static type required by the environment in which the lookup function is used. The type variable `a` will be unified with a type that depends on the application of `lookup`. The encoding of this type (which is indicated by the type class constructor `TC a`) is passed as additional argument to `lookup` such that it can be used in the pattern match of the first function alternative. As a consequence, `lookup dynamiclist + 3` will add `3` to the first dynamic in the list containing an integer value. But, `sinus (lookup dynamiclist)` will take the sinus of the first real value stored in the list of dynamics. So, type dependent functions allow a flexible integration of statically and dynamically typed expressions. The static context can impose restrictions on the dynamic type being demanded.

5.4 Communicating Dynamics

A programmer can store a dynamic to a file in a similar way as he can write a character to a file. It can be done with just one function call. Reading a dynamic from a file can be done in a similar way too.

```
writeDynamic:: Dynamic *File -> *File
// predefined function which appends a dynamic to a file
readDynamic:: *File -> (Bool, Dynamic, *File)
// predefined function which reads a dynamic from a file
sendDynamic:: Dynamic *Channel -> *Channel
// predefined function which sends a dynamic across a channel
```

Dynamics are very useful for the communication between distributed programs. Any data structure or function can via a dynamic be communicated over one and the same channel. A dynamic can be send by applying the function `sendChannel` to the communication channel (which e.g. can be a TCP/IP connection). To receive a dynamic the programmer has to define a callback function that will be applied automatically when the message has been received. These communication primitives are offered by the standard Clean I/O library. The receiving application has to test on the actual type stored in the dynamic as shown in the previous section.

5.5 Implementation

Dynamics are very convenient for the programmer, but hard to implement, in particular in a heterogeneous distributed environment.

First of all one needs a platform independent format for storing and retrieving of dynamics. A dynamic in a program consists of an expression and a decoding of the static type of that expression.

The expression is a term graph which might contain shared sub graphs and which can be cyclic as well. To avoid duplication of work it is important that this sharing is maintained. One can imagine several storage schemes to store a dynamic. One might optimise for compactness or for ease of access. Or one might prefer lazy reading instead of eager reading e.g. when the dynamic to read in is a huge structure. So, although one can store and retrieve a dynamic with just one instruction, several options for doing that need to be given to the programmer.

When a dynamic is read in, the type has to be checked. To check for type consistence, not only the type of the expression itself has to be available. Since user-defined types are possible in the form of algebraic data types, one also has to ensure that the definition of the types in the program and the definitions of the types used in a received dynamic are identical. So, not only the type of the expression, but all type definitions involved have to be stored with it as well. And then, of course, dynamic typing need to be implemented, including dynamic unification and error handling.

But the most complicated things to deal with are caused by the fact that dynamics can contain partially applied functions as well as unevaluated expressions (since Clean is a lazy language). To be able to evaluate a function stored in a dynamic, a program needs the corresponding code. This code should be available in a platform independent format. Fortunately, we have such a format. The Clean compiler generates platform independent abstract machine code, ABC-code. The ABC-code is a kind of byte code, which is compiled by the code generator to native machine code (object code). So, if a dynamic is communicated from one application to another, also the code has to be made available. If the applications run on the same platform, the corresponding object code can be used. Otherwise the ABC-code has to be transmitted and just-in-time compiled to object code. A dynamic linker has been developed (in Clean) which can dynamically link the code to the running receiving application. In this way one can add plug-ins to a running Clean application and use the dynamic type system to test the type correctness.

Finally, another important implementation issue is the version management of dynamics. One can imagine a dynamic stored somewhere in a file in which some function is used. Now, suppose that one discovers a bug in the function definition and repairs it. When the dynamic is now being used one also would like to incorporate the new function definition. However, one can also imagine a complete new version of the software involved. Using the new definition instead of the original one might now become dangerous. Concluding, one needs a version management system to determine which version of the code should be used.

6 Current situation and future developments

Dynamics are part of the new CLEAN system (version 2.0). Although the implementation is not finished yet, most kernel facilities (dynamic type checking, dynamic unification, just-in-time code generation, dynamic linking) have been implemented and work. We believe that the availability of dynamics open a new world of dynamically extending applications. For instance, one can use it to make a kernel operating system, which is initially very small but which grows when new facilities are being used. One can also use it to dynamically repair or modify an application which cannot be stopped. Examples of these are telephone switch systems, airline reservation systems, or the communication software in a satellite. One can use to store the complete status of a program such that one can pick up the work the next day in exactly the state as one left it (persistent programs). Or, one can simply use to store and retrieve the settings of a program with just one function call or fetch a plug-in from the internet.

The new CLEAN system will offer many facilities: sophisticated libraries, which optionally can be included, static linking, dynamic linking, profiling tools, a debugging tool and even a dedicated proof system is under development [19]. To be able to use all these facilities and tools in a user-friendly way, a complete new Integrated Development System has been designed and implemented which makes use of the new I/O library. All the software is written in CLEAN. See our WWW-pages (*www.cs.kun.nl/˜clean*) for the latest news.

References

1. Abadi, M., Cardelli, L., Pierce, B. and Plotkin, G. (1991). Dynamic Typing in a Statically Typed Language. *ACM Transactions on Programming Languages and Systems*, **13**(2):237–268.
2. Achten, P.M., and Plasmeijer, M.J. (1995). The ins and outs of CLEAN I/O. *J. Functional Programming*, **5**(1):81–110.
3. Achten, P.M. (1996). *Interactive Functional Programs - models, methods,and implementations.* Ph.D., University of Nijmegen.
4. Achten, P., and Plasmeijer, R. (1997). Interactive Functional Objects in CLEAN. *In Proc. 1997 Workshop on the Implementation of Functional Languages (IFL'97).* (Hammond, K., Davie, T., and Clack, C. eds.), St.Andrews, Scotland. pp. 387-406. A revised version will appear in the proceedings, LNCS **1467**, Springer Verlag.
5. Armstrong, J., Virding, R., Williams, M. (1993). *Concurrent Programming in* ERLANG. Prentice Hall.
6. Barendregt, H.P. (1984). *The Lambda Calculus - Its Syntax and Semantics* (revised edition). Studies in Logic and the Foundations of Mathematics 103. Elsevier Science Publishers 1984.
7. Barendregt, H.P., Eekelen van, M.C.J.D., Glauert, J.R.W., Kennaway, J.R., Plasmeijer, M.J., and Sleep, M.R. (1987). Term Graph Rewriting. In Bakker, J.W. de, Nijman, A.J., and Treleaven, P.C. eds. *Parallel Architectures and Languages Europe*, Eindhoven, The Netherlands, LNCS **259**, Vol.II. Springer-Verlag, Berlin, pp. 141–158.

8. Barendsen, E., and Smetsers, S. (1996). Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, **6**:579–612.
9. Brus, T., Eekelen, M.C.J.D. van, Leer, M.O. van, and Plasmeijer, M.J. (1987)). CLEAN: A Language for Functional Graph Rewriting. In Kahn, G., ed. *Third International Conference on Functional Programming Languages and Computer Architecture*, Portland, Oregon, USA, LNCS **274**, Springer-Verlag, pp. 364-384.
10. Harper, R., MacQueen, D., and Milner, R. (1986). Standard ML. Edinburgh University, Internal report ECS-LFCS-86-2.
11. Harvey, B. and Wright, M. (1994). *Simply Scheme*. MIT Press.
12. Hoon, W.A.C.A.J. de, Rutten, L.M.W.J., and Eekelen, M.C.J.D. van (1995). Implementing a Functional Spreadsheet in CLEAN. *J. Functional Programming*, **5**(3):383–414, July.
13. Hudak, P., Peyton Jones, S., Wadler, P., *et al.*, (1992). Report on the Programming Language HASKELL. *ACM SigPlan Notices* **27**, (5), pp. 1-164.
14. Girard, J.-Y. (1987). Linear Logic. *Theoretical Computer Science*, **50**: 1–102, 1987.
15. Groningen, J.H.G. van (1996). The Implementation and Efficiency of Arrays in CLEAN 1.1. In Kluge, W., ed. *Implementation of Functional Languages* (Selected papers of 8th International Workshop, IFL96, Bad Godesberg, Germany). LNCS **1268**, pp. 105-124. Springer Verlag.
16. Jones, M.P. (1995). A system of constructor classes: overloading and implicit higher-order polymorphism, *J. Functional Programming*, **5**(1):1–37, January.
17. Leroy, X. (1995). La systeme CAML Special Light: modules et compilation efficace en CAML Research Report 2721, INRIA, November.
18. McCarthy J. (1960). Recursive functions of symbolic expressions and their computation by machine. *Comm ACM*, **4**:184–195.
19. de Mol M. (1998). Clean Prover System. Master Thesis no. 442, University of Nijmegen, 1998.
20. Nöcker, E.G.J.M.H., Smetsers, J.E.W., Eekelen, M.C.J.D. van, and Plasmeijer, M.J. (1991). Concurrent CLEAN. In Aarts, E.H.L., *et al.*, eds, *Parallel Architectures and Languages Europe*, June, Eindhoven, The Netherlands. LNCS **506**, Springer-Verlag, pp. 202–219.
21. Pil, M. (1996). First Class File I/O. In Kluge, W., ed. *Implementation of Functional Languages* (Selected papers of 8th International Workshop, IFL96, Bad Godesberg, Germany). LNCS **1268**, pp. 233-246. Springer Verlag.
22. Plasmeijer, M.J. and van Eekelen, M.C.J.D. (1993). *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley.
23. Plasmeijer, M.J. and van Eekelen, M.C.J.D. (1998). CLEAN 1.3 Language Report. *Technical Report*, *www.cs.kun.nl/˜clean*. Nijmegen.
24. Turner, D.A. (1985). MIRANDA: a non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, Nancy, France (Jouannaud, J.P., ed.), LNCS **201**, pp. 1-16. Berlin: Springer-Verlag.
25. Wadler, Ph. (1997). How to Declare an Imperative. *ACM Computing Surveys*, **29**(3):240-263.
26. Wiering, M., Achten, P., Plasmeijer, R. (1999). Using Clean for Plastform games. In Koopman, P., ed. *Implementation of Functional Languages*, 11th International Workshop, IFL99, Lochem, The Netherlands, Internal Report University of Nijmegen, pp. 144-155. A revised paper will appear in the LNCS series on this workshop.