Graph Rewriting Semantics for Functional Programming Languages

Marko van Eekelen, Sjaak Smetsers, Rinus Plasmeijer
University of Nijmegen

Computing Science Institute, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands,
e-mail marko@cs.kun.nl, sjakie@cs.kun.nl, rinus@cs.kun.nl, fax +31.24.3652525.

Abstract. The lambda calculus forms without any question *the* theoretical backbone of functional programming languages. For the design and implementation of the lazy functional language Concurrent Clean we have used a related computational model: Term Graph Rewriting Systems (TGRS's). This paper wraps up our main conclusions after 10 years of experience with graph rewriting semantics for functional programming languages. TGRS's are not a direct extension of the lambda calculus, so one sometimes has to re-establish known theoretical results. But TGRS's are that much closer to the world of functional programming that its use has been proven to be very worthwhile. In TGRS's functions have names, there are constants, pattern matching and one can choose to either share expressions or copy them.

Graph reduction very accurately models the essential behaviour of most implementations of functional languages and therefore it forms a good base for reasoning about reduction properties as well as the time and space consumption of functional applications.

With uniqueness typing important information can be derived for efficient implementation and for purely functional interfacing with imperative programs.

1 Introduction

There are several models of computation that can be viewed as a theoretical basis for functional programming.

The *lambda calculus* (see Barendregt (1984)), being a well-understood mathematical theory, is traditionally considered to be the purest foundation for modern functional languages like Haskell (Hudak et al. (1990)), ML (Milner (1984)), Miranda (Turner (1985)) and Clean (Brus et al. (1987), Nöcker et al. (1991), Plasmeijer and van Eekelen (1993)).

Later, the concept of *term rewrite systems* (TRS, see Klop (1992)) provided a theory that fits somewhat closer to actual functional programming. Like in functional languages, in TRS's there is a separation between specifications (rule definitions) and applications. Moreover, selection of rewrite rules takes place via pattern matching.

A drawback of both TRS's and lambda calculus is the large gap between those models and the actual implementation of functional languages. E.g. when evaluating an expression of the form FA in λ-calculus or TRS, the argument A might get duplicated many times. This is far from efficient if A contains function applications that still need to be evaluated. Some optimizations of lambda calculus, involving *sharing* of arguments by different functions, have been proposed by Wadsworth (1971) and Lamping (1990).

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 leaks, efficiency, updates, input/output, interfacing with C, and so on. Many of these problems and solutions can be understood better when the semantics are extended incorporating graph structures since all state-of-the-art implementations actually are based on manipulation of 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. Such a model could not only be used to study abstract properties of functional languages, but also as a tool to investigate the practical behaviour of functional programs such as complexity in time and in use of memory.

From the language designer's point of view, it is important to achieve high expressiveness in a language based on a sound semantics resulting in programs that are as efficient as possible. A single framework from theory to implementation is vital to reach this goal. Graph rewriting (Barendregt et al. (1987)) has been proven to serve very well as such a uniform framework:

(i) The *programmer* uses it to reason about the program and to control time and space efficiency of functions and graph structures;

(ii) The *implementor* of the compiler uses it to design optimisations and analysis techniques that are correct with respect to the graph rewriting theory;

(iii) The *theoretician* uses it to build theory for complex analysis of graph rewriting systems used in concrete implementations;

(iv) 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 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 notation chosen in this paper is the one which is most suited to the TGRS-theory (using special symbols and first-order notation assuming a modelling of higher-order via introduction of Apply symbols and rule alternatives). The technical results on basic term graph rewriting are taken from Barendsen and Smetsers (1994); the work on standard typing as well as on uniqueness typing has been presented in Barendsen and Smetsers (1996).

Section 2 of this paper introduces the basic aspects of graph rewriting and its impact on reasoning about the space behaviour of a functional program. Aspects of (distributed) reduction strategies are discussed in section 3. In section 4 conventional typing on graph rewriting rules with algebraic data types is treated. Section 5 gives the rationale behind the introduction of uniqueness typing. In section 6 the core principles are treated of uniqueness typing as an extension of conventional polymorphic typing. Section 7 treats the extension of it with polymorphism also in uniqueness attribute variables. Finally, section 8 contains some concluding remarks.

2 Term graph rewriting

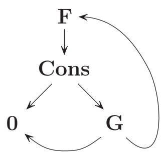
Syntactically, there is not much difference between a program written in Clean and the same program written in some other functional language. Semantically, however, these programs do differ: a functional program in Clean is considered as a set of term graph rewrite rules (TGRS's), and the expression which is going to be evaluated is represented by a computation graph. In fact, term graph rewriting can be seen as an extension of term rewriting with sharing. In term rewrite rules, multiple occurrences of variables lead to duplication of actual instances. In TGRS's, such duplications are avoided by copying references to the objects instead of copying the objects themselves.

Term graph rewrite systems have been introduced in Barendregt et al. (1987), see also Barendsen and Smetsers (1994). The objects in TGRS's are directed graphs in which each node is labelled with a *symbol*. Each symbol $\mathbf{S}$, say, has a fixed arity determining the number of outgoing edges (references to the arguments) of any node labelled with $\mathbf{S}$.

Instead of using drawings, or 4-tuples (see Barendregt et al. (1987) and Barendsen and Smetsers (1994)), we specify graphs in an equational style (cf. Barendregt et al. (1987), Ariola and Klop (1996)). Each equation is a *node specification* of the form

$$n = \mathbf{S}(n_1, \ldots, n_k)$$

where $n, n_1, \ldots, n_k$ are variables. Moreover, the topmost node (*root*) is indicated explicitly. E.g., The graph



can be denoted by

$$\begin{aligned} \langle z \mid \{ z &= \mathbf{F}(c), \\ c &= \mathbf{Cons}(x, g), \\ g &= \mathbf{G}(x, z), \\ x &= \mathbf{0} \qquad\qquad \} \rangle . \end{aligned}$$

Definition 1. A *graph* is a tuple $g = \langle r \mid G \rangle$ where r is a variable, and G a set of equations. The *variable set* of g (notation $\mathrm{V}(g)$) is the collection of variables appearing in r, G. The set of *free variables* of g (notation $\mathrm{FV}(g)$) consists of those in $\mathrm{V}(g)$ that do not appear as the left-hand side of an equation in G; the other (*bound*) variables are indicated by $\mathrm{BV}(g)$. If $\mathrm{FV}(g) = \emptyset$ then g is called *closed*. We will usually identify graphs that only differ in the names of bound variables. The collection of all finite graphs is indicated by $\mathbb{G}$.

The objects on which computations are performed are closed graphs; the others are used for defining graph rewrite rules. These rewrite rules specify possible transformations of graphs.

Definition 2. A *(graph) rewrite rule* $R = \mathbf{Fp} \to g$ consists of left-hand side $\mathbf{Fp}$ (the *pattern*) and a right-hand side g (the *result*). Both $\mathbf{Fp}$ and g are graphs. We have the usual restriction that the unbound variables of g should occur in $\mathbf{p}$. If g is a variable, the R is said to be *collapsing*; otherwise R is *extending*.

To describe the notion of *pattern matching* and of *graph copying*, we introduce structure preserving mappings called *graph homomorphisms*.

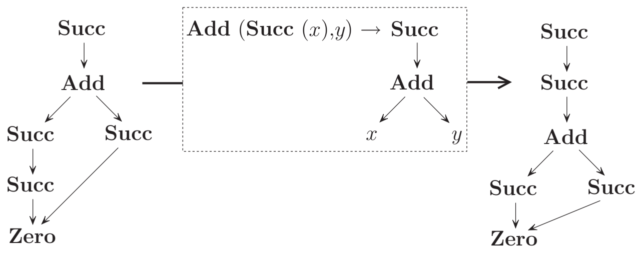
Definition 3. Let g, h be graphs. A function $\varphi : \mathrm{V}(g) \to \mathrm{V}(h)$ is a *(graph) homomorphism* from g to h (notation $\varphi : g \to h$) if for all $n = \mathbf{S}(n_1, \ldots, n_k)$ in g one has

$$\varphi(n) = \mathbf{S}(\varphi(n_1), \ldots, \varphi(n_k))$$

in h. Moreover, we say that φ is *rooted* (notation $\varphi : g \underset{\mathrm{r}}{\to} h$) if $\varphi(r_g) = r_h$ where r_g, r_h are the roots of g, h respectively.

Below we will give an informal definition of graph rewriting. For a full formal description of graph rewriting we refer to Smetsers (1993).

Let R be some rewrite rule. A graph g can be *rewritten* according to R if R is applicable to g, i.e. the pattern of R *matches* g. A *match* μ is a homomorphism from the pattern of R to a subgraph of g. Such a combination $\langle R, \mu \rangle$ is called a *redex*. If a redex has been determined, the graph can be rewritten according to the structure of the right-hand side of the rule involved. This is done in three steps. Firstly, the graph is *extended* with an instance of the right-hand side of the rule. The connections from the new part with the original graph are determined by μ. Then, all references to the root of the redex are *redirected* to the root of the right-hand side. Finally, all 'unreachable nodes' are removed by performing *garbage collection*. The following picture shows the effect of rewriting a graph according to the rule $\mathbf{Add}(\mathbf{Succ}(x), y) \to \mathbf{Succ}(\mathbf{Add}(x, y))$.



We distinguish two kinds of symbols: *function symbols* and *(algebraic) constructor symbols*. Function symbols are symbols for which there exist one or more rewrite rules, whereas algebraic constructors are assumed to be introduced by a so-called *algebraic type system*. An algebraic type system $\mathcal{A}$ contains specifications like

$$\mathsf{List}(\alpha) = \mathbf{Cons}(\alpha, \mathsf{List}(\alpha)) \mid \mathbf{Nil}$$

declaring the data constructors **Cons** and **Nil** (and linking them to the type constructor List).

Obviously, representing data structures as graphs is much more realistic then modeling these by, for instance, λ-terms. By definition, data does not have any computational strength, but apart from that, the ability of using references, sharing and even cycles is very natural when representing data in a functional program.

Definition 4. Let $R = \mathbf{Fp} \rightarrow g$ be a rewrite rule.

(i) R is left-linear if the pattern $\mathbf{Fp}$ is a tree (hence $\mathbf{p}$ does not contain multiple occurrences of the same variable).

(ii) R is a *function/constructor rule* if $\mathbf{p}$ contains only variables and algebraic constructors.

A collection of graphs and a set of rewrite rules can be combined into a graph rewrite system. A special class of so-called functional graph rewrite systems is the subject of further investigations.

Definition 5. (i) A *term graph rewrite system* (TGRS) is a tuple $\mathfrak{T} = \langle \mathcal{G}, \mathcal{R} \rangle$ where $\mathcal{R}$ is a set of rewrite rules, and $\mathcal{G} \subseteq \mathbb{G}$ is a set of closed graphs which is closed under $\mathcal{R}$-reduction (i.e. the reduction relation induced by $\mathcal{R}$).

(ii) Let g be a graph, and let $\langle R_1, \mu_1 \rangle, \langle R_2, \mu_2 \rangle$ be redexes in g such that $R_1 \neq R_2$. These redexes are *overlapping* if $\mu_1(l_1) = \mu_2(l_2)$, where l_1, l_2 are the roots of the left-hand sides of R_1, R_2 respectively.

(iii) A TGRS is *functional* if $\mathcal{R}$ only consists of left-linear function/constructor rules and $\mathcal{G}$ does not contain any graph with overlapping redexes.

For some non-functional TGRS's sharing leads to less choices and hence to less normal forms than the corresponding TRS rules. For functional systems it has been proven that term graph rewriting is adequate for term rewriting, i.e. first lifting a TRS rule to a graph rewriting rule then reducing it in the graph world and then unravel afterwards will yield the same result as performing the reduction entirely within the TRS world. However, functional term graph rewriting itself is not Church-Rosser, as is illustrated by the following example graphs.

The last two graphs are the respective results after contracting the **A**-redex and the **B**-redex in the first graph, using the rules $\mathbf{A}(x) \to x$, $\mathbf{B}(x) \to x$. Clearly, this cannot converge anymore. The problem is induced by so-called *interfering redexes*: redexes involving two different collapsing rewrite rules having each others root as a result. Solving this problem requires a semantic extension which states that self-embedding redexes like $y = \mathbf{A}(y)$ reduce to bottom (a special symbol representing non-termination). Since all implementations of functional languages with lambda-calculus semantics use some kind of graph rewriting scheme they all have hit upon this problem (usually conform lambda calculus semantics as an instance of non-termination) for which it is hard to generate proper code. They all give the programmer a warning that a special situation occurred. This warning can be quite obscure: 'black hole' or close to the graph rewriting semantics: 'cycle in spine'.

Definition 6. (i) Let g be a graph, and let $\langle R_1, \mu_1\rangle, \langle R_2, \mu_2\rangle$ be redexes in g with R_1, R_2 collapsing. These redexes *interfere* if $\mu_1(l_1) = \mu_2(r_2)$ and $\mu_1(r_1) = \mu_2(l_2)$, where l_1, l_2 and r_1, r_2 are the roots of respectively the left- and the right-hand sides of R_1, R_2.

(ii) A TGRS $\mathfrak{T} = \langle \mathcal{G}, \mathcal{R}\rangle$ is *interference-free* if $\mathcal{G}$ does not contain any graph with interfering redexes.

The following result has been proven in Barendsen and Smetsers (1993)

Theorem 7. *Let $\mathfrak{T} = \langle \mathcal{G}, \mathcal{R}\rangle$ be functional and interference-free. Then reduction according to $\mathcal{R}$ satisfies the Church-Rosser property. In a picture:*

$$\begin{tikzcd}
 & \cdot \arrow[dl, "\mathcal{R}"'] \arrow[dr, "\mathcal{R}"] & \\
\cdot \arrow[dr, dotted, "\mathcal{R}"] & & \cdot \arrow[dl, dotted, "\mathcal{R}"'] \\
 & \cdot &
\end{tikzcd}$$

2.1 Programming with sharing semantics

In functional programming languages it is folklore to consider multiple occurrences of variables and let definitions as instructions for the system to share the corresponding objects as much a possible.

In Clean, reasoning about programs is reasoning about graphs and graph rewrite rules. It is straightforward to denote shared computations and cyclic structures. For instance, the semantics of Clean prescribes that the argument n in the function **Double**

$$\mathbf{Double}(n) \to +(n, n)$$

is shared in the graph constructed on the right-hand side.

Lambda calculus takes a drastic decision in making no difference between functions and data. Clearly, this allows a programmer to use functions as first class citizens. However, the ability to reason about the memory consumption of the data structures is lost. Graph rewrite semantics can give this ability (still having functions as first class citizens) by providing for each construct and each elementary value the node size. The size of the graph representing the data structure is then easily calculated. Sharing semantics gives a uniform framework to give meaning to language constructs such as structure matching (indicating a sub-graph on the left-hand-side where the pattern is not just a variable), cycles (not a magic system feature but explicitly defined), Combinatory Applicative Forms (parameterless rewrite rules that are considered as global graphs and therefore evaluated at most once).

Not only multiple occurrences of argument variables denote sharing but also local definitions can denote sharing. In Clean for local definitions the choice of sharing can be done explicitly by indicating that a local definition must be a graph or must be a function depending on whether it is wanted to trade space against time. When no explicit choice is made an analysis is performed to decide whether it is possible to make it a graph or whether it has to be a function due to the internal use of variables of an outside scope.

A nice example of the use of cycles is the following solution of the hamming problem that employs a cycle to change the order of complexity from exponential into a polynomial one. The function **Mer** for merging two possibly infinite lists is defined employing structure matching in the left-hand side.

$$\mathbf{Ham} \rightarrow y : \mathbf{Cons}(1, \mathbf{Mee}(\mathbf{Mer}(\mathbf{Map}(*(2), y), \mathbf{Map}(*(3), y)), \mathbf{Map}(*(5), y)))$$

Space Leaks

A programmer can be confronted with unexpected run-time increase of memory consumption: a so-called *space leak*. Due to the lack of graph semantics for many languages, for many functional programmers a space leak has become a mythical problem. Due to the absence of any handle at all to address space consumption in lambda calculus semantics, it is even impossible to express the following main differences in causes of space leaks.

A space leak can have three causes: a design error, a programming error or a system error. *Design errors* typically lead to space leaks when some kind of log is kept internally in the program. This log will consume more and more space. Such kind of errors can occur in a functional language just as easy as in a non-functional language. *Programming errors* can lead to a space leak when unintentionally the program refers to parts of the graph that are not needed anymore. Such errors are much more often made by a functional programmer than by a non-functional programmer. The reason for that is twofold: firstly, a non- functional programmer has to allocate and deallocate each memory cell explicitly and secondly it is hard for a functional programmer to keep track of evaluation of subgraphs due to lazy evaluation. A satisfactory solution is to allow the programmer to make data structures and arguments strict when he is sure that that is the intention. Theoretically this may lead to unwanted non-termination in which case however the programmer has made an error of another kind: run-away recursion. *System errors* leading to a space leak can occur when the implementor of the compiler optimises deviating from the graph rewrite semantics. Typically this is a graph which is still held on the stack when this should not have been done. Usually it has to do with explicit or implicit selection functions for local tuple or record definitions. When the corresponding subgraphs occur both in a lazy as well as in a strict context the situation can be quite complicated. Anyhow, such errors are system implementation errors which must be fixed by the systems implementor.

2.2 Implementation optimisations

The basic scheme which is implemented will always be the standard graph rewriting semantics. Deviations are allowed if the result is still in conformity with these semantics. Optimisations such as garbage collection, the use of stacks, overwriting of nodes, performing calculations on a special value stack etc. are all validated against the graph rewriting semantics (Barendregt et al. (1987), Smetsers et al. (1991)).

3 Reduction Strategies

When using term rewriting systems or term graph rewriting systems as a model of computation, these systems are usually equipped with a *reduction strategy*.

In order to determine *redexes* to be contracted one often uses an auxiliary function selecting *nodes* that are intended to be rewritten in the subject graph. For ordinary functional TGRS's such a node uniquely determines a redex.

3.1 Strictness analysis

For strictness analysis pattern information is very valuable since arguments have to be (partly) evaluated to perform the pattern match. Incorporating this information in the computational model makes it possible to define strictness analysis based on abstract reduction with very good results (Nöcker (1993)). The distinction between data and functions gives the opportunity to define data structures that are inherently strict (e.g. a complex number consisting of two reals which always have to be evaluated both) (Nöcker and Smetsers (1993)).

3.2 Distributed evaluation

Although pure TGRS's are convenient to model *sequential* computations, they do not capture some aspects that are important in *parallel* implementations.

In a functional program the evaluation result of any expression is independent of the chosen reduction order by the absence of side effects. This makes functional languages attractive for implementation on parallel hardware. An important class of parallel machines is formed by systems of loosely coupled processors, each equipped with its own local memory. An implementation of functional languages on such a machine has to deal with the exchange of data during program execution. This is a nontrivial matter. In the literature, however, the communication mechanism is often considered as a pure implementation issue and therefore handled *ad hoc*.

In the 'overall view' of a computation, the communication between two processors involves duplication of data. In order to investigate theoretical properties of such a computation, graph rewriting systems will be extended with a general copying mechanism. The information transfer is determined by the moment a processor needs data, and by the amount of information that is available (i.e. not subject to a present computation) at that moment. In terms of copying this means that it should be possible to defer copying at some points in the subject graph until these parts of the computation have been finished. This way of copying is called *lazy copying*. Lazy copying can be used to model communication *channels*, as proposed in Eekelen et al. (1991). Together with a mechanism for creating parallel reduction processes, the lazy copying formalism provides a specification method for arbitrary process topologies with various kinds of communication links.

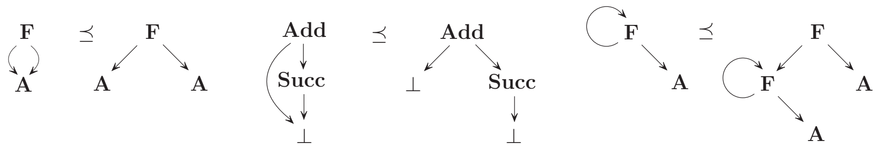
As a first step towards a full description of the lazy copying mechanism, we extend graph rewriting with a simple notion of graph copying and show that the combination of graph copying and ordinary reduction is sound.(See also Barendsen and Smetsers (1993).)

Definition 8. The ordering $\preceq$ on $\mathbb{G}$ is defined as follows

$$g \preceq h \quad \text{if} \quad \varphi : h \underset{r}{\rightarrow} g \text{ for some } \varphi.$$

Here h can be seen as the result of partially *unraveling* g, or, more operationally, h can be obtained from g by duplicating some of g's nodes.

Example.



To each graph g one can associate a (possibly infinite) tree $\mathbb{U}(g)$ by an operation called *complete unraveling*. By $g \sim_{\mathrm{T}} h$ we denote that g and h are *tree equivalent*, i.e. have the same unraveling. The following result states that if two graphs are tree equivalent the have a common finite unraveling.

Proposition 9. *Let $g \sim_{\mathrm{T}} g'$. Then there exists a graph $h \in \mathbb{G}$ such that $g \preceq h$ and $g' \preceq h$.*

Unraveling a graph might influence the redexes of that graph. The following result implies that unraveling and ordinary reduction can be combined preserving the Church-Rosser property. In order to describe the combination of unraveling and ordinary reduction we denote unraveling as a rewrite step, i.e. $g \underset{\preceq}{\rightarrow} h$ if $g \preceq h$.

Theorem 10. *Let $\mathfrak{T} = \langle \mathcal{G}, \mathcal{R} \rangle$ be functional and interference-free. Then $\mathcal{R}_{\preceq}$-reduction is Church-Rosser.*

Towards lazy copying

In C-TGRS's (TGRS's extended with copying) a copying action is considered as a rewrite step. The copy redexes are indicated by special *copy nodes*: nodes containing the (predefined) symbol **C** of arity 1, pointing to the graph to be copied. However, in order to obtain a refined control over the distribution of both work and data, it is not sufficient anymore if the reduction strategy would indicate copy redexes solely by their root node. It is also necessary to specify which parts of the indicated subgraphs are really copied and which parts remain shared (remember that copying is partial unraveling). To have some control over the choice of these copied segments, the language of graph rewriting has been extended with the possibility of attaching a so called *defer* attribute to nodes. Intuitively, this attribute (temporarily) prevents the node in question of being copied. Defer attributes are also allowed to appear in the right-hand side of a rewrite rule. In Eekelen et al. (1991) these attributes are used to define primitives for specifying communication for various kinds of parallelism, varying from simple divide-and-conquer parallelism to complicated, non-hierarchical process structures.

4 Conventional Typing

Adding type information to a program is important for several reasons. Types enhance readability; requiring correctness of types removes many programming errors and the type information is vital for the efficiency of the generated code.

We will introduce a notion of type assignment which is not defined using a set of deduction rules but, more directly, by supplying all the nodes of the computation graph and the rewrite rules with types. Correctness of such a type assignment is formulated in terms of local requirements for each of these nodes.

Type assignment has the following two properties. The first one is known as the *subject reduction property* indicating that typability is preserved during reduction. Among other things, this will ensure that source-to-source transformations of programs do not lead to untypability. The second one is the so-called *principal type property*: if a graph (or a function) is typable, then a most general typing (i.e. a typing of which all other typings are instances) can be computed effectively.

This notion of conventional typing is common in most functional programming languages. It combines simple Curry typing with an instantiation mechanism to deal with different occurrences of function and constructor symbols. This weak form of polymorphism is necessary due to the separation of specifications (function definitions, algebraic types) from applications.

Definition 11. *Types* are built up from type variables and type constructors.

$$\sigma \ ::= \ \alpha \mid \mathsf{T}\boldsymbol{\sigma} \mid \sigma_1 \to \sigma_2 .$$

Here, T ranges over type constructors which are assumed to be introduced by an algebraic type system $\mathcal{A}$.

The function space type constructor $\rightarrow$ is used when dealing with higher-order functions. The typing system presented in this overview, however, is first order; for a description of the full higher-order system see Barendsen and Smetsers (1996).

We first associate a type with the constructors and function symbols. The notion of type assignment for graphs is parametric in the choice of these symbol types.

Definition 12. A *symbol type* of arity k is a $k+1$ tuple $(\sigma_1, \ldots, \sigma_k, \tau)$. We will suggestively denote this as

$$(\sigma_1, \ldots, \sigma_k) \rightarrowtail \tau.$$

The types $\boldsymbol{\sigma}$ are called *argument types* and τ is the *result type*.

Definition 13. (i) Let $\mathcal{A}$ be an algebraic type system. The specifications in $\mathcal{A}$ give the types of the algebraic constructors. Let

$$\mathsf{T}\boldsymbol{\alpha} = \mathbf{C}_1\boldsymbol{\sigma_1}| \cdots$$

be the specification of T in $\mathcal{A}$. Then we write

$$\mathcal{A} \vdash \mathbf{C}_i : \boldsymbol{\sigma_i} \rightarrowtail \mathsf{T}\boldsymbol{\alpha}.$$

For example, for lists one has

$$\mathcal{A} \vdash \mathbf{Nil} : \mathsf{List}(\alpha), \qquad \mathcal{A} \vdash \mathbf{Cons} : (\alpha, \mathsf{List}(\alpha)) \rightarrowtail \mathsf{List}(\alpha).$$

(ii) The *function* symbols are supplied with a type by a *function type environment* $\mathcal{F}$, containing declarations of the form

$$\mathbf{F} : (\sigma_1, \ldots, \sigma_k) \rightarrowtail \tau,$$

where k is the arity of $\mathbf{F}$. In this case we write

$$\mathcal{F} \vdash \mathbf{F} : \boldsymbol{\sigma} \rightarrowtail \tau.$$

(iii) The symbol types obtained so far are referred to as the *standard types* (in $\mathcal{F}, \mathcal{A}$) of the symbols. These are regarded as type *schemes*: other types are obtained by instantiation, using the following rule ($[\alpha := \rho]$ denotes substitution).

$$\boxed{\dfrac{\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \boldsymbol{\sigma} \rightarrowtail \tau}{\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \boldsymbol{\sigma}[\alpha := \rho] \rightarrowtail \tau[\alpha := \rho]}}$$

For the sequel, fix an algebraic system $\mathcal{A}$ and a function type environment $\mathcal{F}$. Now we can develop a system of type assignment for graphs and for graph rewrite rules.

Definition 14 Graph typing. Let $g = \langle r \mid G \rangle$ be a graph. A *typing* for g is a function $\mathcal{T}$ assigning a type to each element of $\mathrm{V}(g)$ such that for any equation $x = \mathbf{S}(\mathbf{y})$ in G

$$\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \mathcal{T}(\mathbf{y}) \rightarrowtail \mathcal{T}(x).$$

That is, $\mathcal{T}$ is a type assignment to nodes that satisfies 'local correctness': the actual argument types of each symbol application should be instances of the corresponding formal argument types, as specified by $\mathcal{F}, \mathcal{A}$.

Note that this definition with the three definitions above are all that is required for defining conventional typing on graph rewrite rules using algebraic data types.

A typing for a graph is called *principal* if all other typings can be obtained via instantiation. The following result is proved in Barendsen and Smetsers (1996)

Principal Typing Theorem 15 *There exists a recursive function* Type *such that*

$$\begin{aligned} g \textit{ is typable} &\Rightarrow \mathrm{Type}(g) \textit{ is a principal typing for } g; \\ g \textit{ is not typable} &\Rightarrow \mathrm{Type}(g) = \mathsf{fail}. \end{aligned}$$

Typing of graph rewrite rules can be formulated as follows.

Definition 16 Rule typing. (i) A type assignment $\mathcal{T}$ to variables can be extended to algebraic *patterns* in a straightforward way:

$$\mathcal{F}, \mathcal{A} \vdash \mathbf{C} : \boldsymbol{\rho} \rightarrowtail \sigma,\ \mathcal{T}(\mathbf{p}) = \boldsymbol{\rho} \ \Rightarrow\ \mathcal{T}(\mathbf{Cp}) = \sigma.$$

(ii) Say the standard type of $\mathbf{F}$ in $\mathcal{F}$ is $\boldsymbol{\sigma} \rightarrowtail \tau$. Then the rewrite rule $\mathbf{Fp} \to g$ is *type correct* if for some $\mathcal{T}$ one has

$$\begin{aligned} \mathcal{T}(\mathbf{p}) &= \boldsymbol{\sigma}, \\ \mathcal{T}(g) &= \tau. \end{aligned}$$

Note that the type assignment to the pattern $\mathbf{p}$ is uniquely determined by the input types $\boldsymbol{\sigma}$: given $\mathbf{C}$ and σ, there is at most one sequence $\boldsymbol{\rho}$ such that $\mathcal{F}, \mathcal{A} \vdash \mathbf{C} : \boldsymbol{\rho} \rightarrowtail \sigma$.

(iii) A collection of rewrite rules is *type correct* if every member is.

Typing is preserved during reduction (the so-called *subject reduction property*).

Theorem 17 Subject Reduction. *Suppose $\mathcal{R}$ is type correct. Then*

$$\left.\begin{aligned} g &: \sigma \\ g &\twoheadrightarrow_{\mathcal{R}} g' \end{aligned}\right\} \Rightarrow g' : \sigma.$$

5 Introduction to Uniqueness Typing

In recent years, various proposals have been brought up to capture the notion of assignment in a functional context. This desire is paradoxical, because the absence of side effects is one of the main reasons why functional programming languages are often praised. As a consequence of this absence, functional languages have the fundamental property of *referential transparency*: each (sub)expression denotes a fixed value, independently of the way this value is computed.

However, if, for instance, a functional program and the real world have to be linked this can only be done by identifying the physical object in the program by a structure that indicates the state of the object: e.g. a lamp is on or off. In the program this is represented by a constant lamp_on which can be True or False. This link is direct. The value of the constant always corresponds to the state of the object. Now suppose that there are functions to change the state of the object (switch the lamp on or off) and to inspect the state of the object (is the lamp on or off?). The functions can be seen as a write function and a read function. However, this induces a problem. Since it is possible to duplicate references to an object, at several places in a program a function to switch the lamp on or off can occur. The result will depend on the order in which the function calls are evaluated. Clearly, referential transparency will not hold anymore. If duplicating the reference would mean duplicating the physical object there would be no problem at all (e.g. the operating system VAX/CMS did that with files: when editing copies were created automatically). If the physical object cannot be duplicated (or if the interface requires that the object is not duplicated) the solution in a pure functional language one way or the other must guarantee that at most one reference to such an object is made.

In this paper, we regard assignments in a broad sense: these include direct mutation of memory contents but also more indirect I/O operations like file manipulations. The common aspect of such operations is their *destructive* behaviour: they (irreversibly) change the state of their input objects.

One solution for this problem can be achieved without any extensions of the functional framework: by delivering a sequence of instructions for the operating system as a result of a functional expression. One could call this a *delegating* approach, since the computation only *prepares* for the external execution of, for example, I/O tasks. In the literature, this method is known as *stream based* I/O. An application can be found in the languages *Miranda* and *Haskell*.

Rather than an indirect treatment of destructive operations one would like to incorporate these operations (and hence the objects they operate on) directly in a functional programming language. This admits a more refined control of files, for example. However, by admitting them without precaution one looses referential transparency. If two destructive functions operate on the same file, for example, the result of the program depends on the order in which these operations are performed.

The problem is, therefore, to identify suitable restrictions on the usage of destructive operations. The essence of recent solutions (e.g., Wadler (1990), Guzmán and Hudak (1990)) is to restrict destructive operations to arguments that are accessed only once. Syntactically, this boils down to restricting the number of occurrences of these arguments inside each program to one.

The uniqueness type system for graph rewrite systems presented in Barendsen and Smetsers (1993). offers the possibility to indicate such reference count requirements in type specifications of functions. This is done via so-called *uniqueness types* which are annotated versions of traditional Curry-like types. E.g. the operation **WriteChar** which writes a character to a file is typed with **WriteChar** : $(\mathsf{Char}^{\times}, \mathsf{File}^{\bullet}) \to \mathsf{File}^{\bullet}$. Here, $\bullet, \times$ stand for 'unique' (the requirement that the reference count is 1) and 'non-unique' (no reference requirements) respectively.

Although this analysis is primarily intended for inherently destructive operations it can also help to improve storage management. Consider, for instance, the following list reversing function which can be implemented efficiently as a 'destructive' function if the given uniqueness type is used.

$$\begin{array}{l} \mathbf{Rev} : \mathsf{List}^{\bullet}(\alpha^{\times}) \rightarrowtail \mathsf{List}^{\bullet}(\alpha^{\times}) \\ \mathbf{Rev}\ (l) \quad \to \quad \mathbf{H}(l, \mathbf{Nil}) \end{array}$$

$$\begin{array}{l} \mathbf{H} : (\mathsf{List}^{\bullet}(\alpha^{\times}), \mathsf{List}^{\bullet}(\alpha^{\times})) \rightarrowtail \mathsf{List}^{\bullet}(\alpha^{\times}) \\ \mathbf{H}\ (\mathbf{Nil}, \ell) \qquad\qquad \to \ell \\ \mathbf{H}\ (\mathbf{Cons}(h,t), \ell) \to \mathbf{H}(t, \mathbf{Cons}(h, \ell)) \end{array}$$

Note that **H**'s first argument has reference count 1 in any type correct application. The topmost node of this argument is not used in the result of the function. Hence this node becomes obsolete and can be re-cycled: not only its space but also also parts of its contents. In fact it already suffices to change the reference to t to point to ℓ_2. Such re-usage of space is often called *compile-time garbage collection*.

The idea of restricting occurrences of input objects by a type system is not new: we can make use of ideas developed in so-called resource conscious logics like linear logic. Via the propositions-as-types correspondence (relating inputs to assumptions and functions to proofs) restrictions on usage of assumptions in these logics gives the desired reference count limitations.

Since we deal with both destructive and harmless operations, a purely linear-like typing system (in which neither copying nor discarding of input is allowed) is too restrictive for our purposes. Instead, we propose to divide the type system into two layers: a 'resource conscious' part in which occurrences are limited, and a 'conventional' part with no reference restrictions. The two layers are connected: it is possible to move from the $\bullet$ layer to the $\times$ layer. These transitions are regulated by the type system: we have (1) a subtype relation allowing a unique object to be seen as a conventional one (in case the accessing function has no reference requirements) and (2) a type correction mechanism (forcing an object with reference count greater than 1 to be regarded as non-unique).

We will now explain the graph-theoretic intuition of our type system. We have seen that the environment type $\mathbf{F} : \sigma^{\bullet} \rightarrowtail \cdots$ specifies that in any application the concrete function argument should have reference count 1. In the same way, uniqueness of results is specified: if $\mathbf{G} : \cdots \rightarrowtail \sigma^\bullet$, then a well-typed expression $\mathbf{F}(\mathbf{G}(E))$ remains type-correct, even if $\mathbf{G}(E)$ is subject to computation.

The above-mentioned transitions between the type layers are motivated as follows. Sometimes, uniqueness is not required. If $\mathbf{F} : \sigma^\times \rightarrowtail \cdots$ then still $\mathbf{F}(\mathbf{G}(E))$ is type correct. This is expressed in the subtype relation $\leq$, such that roughly $\sigma^\bullet \leq \sigma^\times$. Offering a non-unique argument if a function requires a unique one fails: $\sigma^\times \nleq \sigma^\bullet$. The subtype relation is defined in terms of the ordering $\bullet \leq \times$ on attributes. In an application an argument can also be non-unique if it has reference count greater than 1 (even though the type of the argument expression itself is unique). This is covered by a correction mechanism: a unique result may be used more than once, as long as only non-unique supertypes are required.

From the types given above it can be seen that the layers $\bullet$ and $\times$ are have some internal structure induced by the presence of type constructors: a $\bullet$ type (at the outermost level) can have parts marked with $\times$ (and vice versa). This fine structure, with the possibility of specifying linear or conventional behaviour of substructures, is a powerful feature of our system.

Pattern matching causes a function to have access to 'deeper' arguments via *data paths* instead of a single reference. This gives rise to 'indirect sharing' of objects by access via intermediate data nodes. (These data nodes that are used to connect the individual parts of a compound object are often called the *spine* of that object.) For example, if a function **F** has access to a list with non-unique spine, the list elements should also be considered as non-unique for **F**: other functions may access them via the spine. This effect is taken into account by a restriction on the uniqueness types of data constructors: the result of a constructor is unique whenever one of its arguments is. For the constructor **Cons** of lists, for example, the possible uniqueness variant are:

$$\begin{array}{l}
\mathbf{Cons} : (\alpha^\times, \mathsf{List}^\times(\alpha^\times)) \rightarrowtail \mathsf{List}^\times(\alpha^\times)\ (1)\\
\mathbf{Cons} : (\alpha^\times, \mathsf{List}^\bullet(\alpha^\times)) \rightarrowtail \mathsf{List}^\bullet(\alpha^\times)\ (2)\\
\mathbf{Cons} : (\alpha^\bullet, \mathsf{List}^\bullet(\alpha^\bullet)) \rightarrowtail \mathsf{List}^\bullet(\alpha^\bullet)\ (3)
\end{array}$$

With (1) ordinary lists can be built. (2) can be used for lists of which the 'spine' is unique, and (3) for lists of which both the spine and the elements are unique. Observe that in the above example, the List argument of **Cons** is always attributed in the same way as the corresponding List result. In general, such a uniform way of attributing recursive occurrences of a type constructor leads to *homogeneous* data objects: All recursive parts of such an object have the same uniqueness properties as the object itself.

We can also express propagation by using the $\leq$ relation. E.g.

$$\mathbf{Cons} : (\alpha^u, \mathsf{List}^v(\alpha^u)) \rightarrowtail \mathsf{List}^v(\alpha^u)$$

is well-attributed if $v \leq u$. Note that this indeed excludes a constructor for $\mathsf{List}^\times(\mathsf{Int}^\bullet)$.

Some parts of the uniqueness type system are complicated. The treatment of cyclic dependencies is subtle; moreover dealing with higher-order functions is a non-trivial matter. This seems to be a common aspect of related approaches.

The way references are counted can be refined, by making use of information on the evaluation order. To avoid unnecessary complications we will not treat this in detail but give an idea of the method at the end of this section.

For reasons of clarity, we have split the presentation of the uniqueness typing into two parts. After a brief introduction, a simple (monomorphic) uniqueness type system is given. Then, uniqueness polymorphism is added to the system, e.g. to enable the determination of principal types. Again, both systems will be first order.

6 Simple Uniqueness Typing

Algebraic Uniqueness Types

Uniqueness types are constructed from conventional types by assigning a uniqueness attribute to each subtype. We will denote the attributes as superscripts; for non-variable types these are attached to the topmost type constructor of each type. Below, $S, T, \ldots$ range over uniqueness types and $u, v, \ldots$ over the attributes $\bullet, \times$. The outermost attribute of S is denoted by $\ulcorner S \urcorner$. Moreover $|S|$ denotes its underlying conventional type.

Definition 18. The *subtype relation* $\leq$ is very simple: the validity of $S \leq S'$ depends subtypewise on the validity of $u \leq u'$ with u, u' attributes in S, S'. One has, for example,

$$\mathsf{List}^u(\mathsf{List}^v(\mathsf{Int}^w)) \leq \mathsf{List}^{u'}(\mathsf{List}^{v'}(\mathsf{Int}^{w'})) \quad \text{iff} \quad u \leq u', v \leq v', w \leq w'.$$

In order to account for multiple references to the same object (sharing) we introduce a uniqueness correction.

Definition 19. For each S, we construct the smallest non-unique supertype $[S]$ of S, as follows.

$$[\alpha^u] = \alpha^\times,$$
$$[\mathsf{T}^u \mathbf{S}] = \mathsf{T}^\times \mathbf{S}.$$

The last clause possibly introduces types like $\mathsf{List}^\times(\mathsf{Int}^\bullet)$. Contrasting Turner et al. (1995), we allow these types in our system. This is harmless since these 'inconsistent' types have no proper inhabitants (for example, there is no **Cons** yielding type $\mathsf{List}^\times(\mathsf{Int}^\bullet)$).

The notion of standard type is adapted in the following way. As can be seen from the List example, there are several standard types for each data constructor.

Definition 20. (i) As before, *standard types* of function symbols ($\mathbf{F} : \mathbf{S} \rightarrowtail T$) are collected in an environment $\mathcal{F}$.

(ii) Say the algebraic environment $\mathcal{A}$ contains

$$\mathsf{T}\boldsymbol{\alpha} = \mathbf{C}_1\boldsymbol{\sigma_1} | \cdots$$

A set of *standard types* for $\mathbf{C}_i$ consists of attributed versions of the conventional type $\boldsymbol{\sigma_i} \rightarrowtail \mathsf{T}\boldsymbol{\alpha}$, such that

(1) multiple occurrences of the same variable and of the constructor T have the same uniqueness attribute throughout each version;

(2) each version is uniqueness propagating;

(3) the set contains at most one version for each attributed variant of $\mathsf{T}\boldsymbol{\alpha}$.

For these standard types $\mathbf{S} \rightarrowtail T$ we set $\mathcal{A} \vdash \mathbf{C} : \mathbf{S} \rightarrowtail T$ as before.

Definition 21. Symbol types are *instantiated* via the rule

$$\boxed{\frac{\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \mathbf{S} \rightarrowtail T \qquad \ulcorner \alpha \urcorner = \ulcorner R \urcorner}{\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \mathbf{S}[\alpha := R] \rightarrowtail T[\alpha := R]} \text{(instantiation)}}$$

Now we are ready to present a notion of uniqueness typing for graphs, based on type assignment to nodes in graphs.

For a uniform treatment of all nodes in the graph, we will introduce the following notion.

Definition 22. (i) Let g be a graph. Then g^+ is the graph that results from g by adding a new root r^+ with in-degree 0 containing the data symbol **Root** of arity 1, pointing to the root r of g.

(ii) The standard types of **Root** are given by $\alpha^u \rightarrowtail \alpha^u$.

Since **Root** is not a function, the root of g^+ will never be involved in the rewrite process. Furthermore, for each cycle in g^+ there is always an external reference (i.e., a reference from a node that is not part of the cycle) to that cycle.

Definition 23. Let n be a node in g. The *reference count* of n in g (notation $\mathrm{rc}_g(n)$ or just $\mathrm{rc}(n)$) is $\otimes$ if n appears more than once in the right-hand sides of equations in g^+, and $\odot$ otherwise. Note that this mechanism only differs from ordinary reference counting at the root of the graph, notably when the root is part of a cycle.

Definition 24 Uniqueness graph typing. A *uniqueness typing* for a graph g is a function $\mathcal{T}$ assigning a uniqueness type to each node in g^+ such that for any node specification $x = \mathbf{S}(\mathbf{y})$ there exist types $\mathbf{S}$ with the following properties.

$$\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \mathbf{S} \rightarrowtail \mathcal{T}(x),$$

and for all $i \leq k$

$$\begin{aligned} \mathcal{T}(n_i) &\leq S_i \;\text{ if } \mathrm{rc}_g(n_i) = \odot, \\ [\mathcal{T}(n_i)] &\leq S_i \;\text{ if } \mathrm{rc}_g(n_i) = \otimes. \end{aligned}$$

We say that $\mathcal{T}$ *types* g with S (notation $\mathcal{T}(g) = S$) if moreover $\mathcal{T}(r^+) = S$.

The constraints in the above definition reflect the uniqueness property of function applications mentioned in Section 5. If, say, $\mathbf{F}$ with arity 2 has a standard type in which the first argument is unique, then for any application $x = \mathbf{F}(y, z)$ in a type correct graph g we have that $\mathrm{rc}_g(y) = 1$.

We now relate function typings with typings of graph rewrite rules. Uniqueness typing of graph rewrite rules is defined as follows.

Definition 25 Rule uniqueness typing. (i) A uniqueness type assignment $\mathcal{T}$ to variables can be extended to patterns in the following way:

$$\mathcal{F}, \mathcal{A} \vdash \mathbf{C} : \mathbf{S} \rightarrowtail T,\ \mathcal{T}(\mathbf{p}) = \mathbf{S} \ \Rightarrow\ \mathcal{T}(\mathbf{C}\ \mathbf{p}) = T.$$

(ii) Say the standard type for $\mathbf{F}$ in $\mathcal{F}$ is $\mathbf{S} \rightarrowtail T$. Then the rewrite rule $\mathbf{Fp} \rightarrow g$ is *(uniqueness) type correct* if for some uniqueness typing $\mathcal{T}$ one has

$$\begin{aligned} \mathcal{T}(\mathbf{p}) &= \mathbf{S}, \\ \mathcal{T}(g) &= T. \end{aligned}$$

(iii) A collection of rewrite rules is type correct in $\mathcal{F}$ if every member is.

Once more, we have that typing is preserved during reduction.

Theorem 26 Uniqueness Subject Reduction. *Suppose $\mathcal{R}$ is uniqueness type correct. Then for any g, h, S*

$$\left.\begin{array}{r} g : S \\ g \twoheadrightarrow_{\mathcal{R}} h \end{array}\right\} \Rightarrow\ h : S.$$

Moreover the latter type assignment coincides with the original for g with respect to the free variables of h.

Proof. See Barendsen and Smetsers (1993). □

7 Polymorphic Uniqueness Typing

In order to denote uniqueness schemes, we extend the attribute set with *attribute variables* $(a, b, a_1, \ldots)$. This increases the expressiveness of the type system. Moreover, attribute polymorphism is needed for the determination of 'principal' uniqueness variants of typings.

Uniqueness constraints are indicated by (finite) sets of attribute inequalities called *attribute environments*. For example, the standard type of the symbol **Cons** is now expressed by

$$\mathbf{Cons} : (\alpha^a, \mathsf{List}^b(\alpha^a)) \rightarrowtail \mathsf{List}^b(\alpha^a) \quad | \quad b \leq a.$$

Note that this expression captures the collection of standard types for **Cons** in one single type. The former types for **Cons** can be obtained by substituting concrete attributes for a and b satisfying the requirement $a \leq b$. The same is done for all symbols: each symbol has one polymorphic standard type $\mathbf{S} \rightarrowtail T \,|\, \Gamma$.

Some of the notions of the previous section (type environment, subtyping) are re-defined relative to attribute environments.

Definition 27. (i) As to the attribute relation $\leq$, we say that $u \leq v$ is *derivable* from the attribute environment Γ (notation $\Gamma \vdash u \leq v$) if $\Gamma \vdash u \leq v$ can be produced by the axioms

$$\Gamma \vdash u \leq v \qquad \text{if } (u \leq v) \in \Gamma,$$
$$\Gamma \vdash u \leq u, \qquad \Gamma \vdash u \leq \times, \qquad \Gamma \vdash \bullet \leq u$$

and rule

$$\frac{\Gamma \vdash u \leq v \quad \Gamma \vdash v \leq w}{\Gamma \vdash u \leq w}.$$

(ii) This denotation is extended to finite sets of inequalities: $\Gamma \vdash \Gamma'$ if $\Gamma \vdash u \leq v$ for each $(u \leq v) \in \Gamma'$. By $u = v$ we denote the pair $u \leq v, v \leq u$.
(iii) We say that Γ is *consistent* if $\Gamma \nvdash \times \leq \bullet$.

Definition 28. For every Γ, the validity of the subtyping relation $\leq_\Gamma$ in $S \leq_\Gamma S'$ again depends subtypewise on the validity of $\Gamma \vdash u \leq u'$ with u, u' attributes in S, S'. One has, for example,

$$\mathsf{List}^u(\mathsf{List}^v(\alpha^w)) \leq_\Gamma \mathsf{List}^{u'}(\mathsf{List}^{v'}(\alpha^{w'})) \quad \text{iff} \quad \Gamma \vdash u \leq u', v \leq v', w \leq w'.$$

Definition 29. The instantiation rules for the polymorphic system are

$$\frac{\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \mathbf{S} \rightarrowtail T \mid \Gamma \qquad \Gamma' \vdash \Gamma}{\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \mathbf{S} \rightarrowtail T \mid \Gamma'} \text{(attribute instantiation)}$$

$$\frac{\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \mathbf{S} \rightarrowtail T \mid \Gamma \qquad \Gamma \vdash \ulcorner \alpha \urcorner = \ulcorner R \urcorner}{\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \mathbf{S}[\alpha := R] \rightarrowtail T[\alpha := R] \mid \Gamma} \text{(instantiation)}$$

Reformulation of uniqueness graph typing leads to the following.

Definition 30 Polymorphic uniqueness graph typing. Let $g = \langle r \mid G \rangle$ be a graph. A *polymorphic uniqueness typing* for g is a pair $\langle \mathcal{T}, \Gamma \rangle$ where $\mathcal{T}$ is a uniqueness type assignment to nodes in g and Γ is a consistent coercion environment such that for any equation $x = \mathbf{S}(\mathbf{y})$ in G there exist types $\mathbf{S}$ with the following properties.

$$\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \mathbf{S} \rightarrowtail \mathcal{T}(x) \mid \Gamma,$$

and for all $i \leq k$

$$\mathcal{T}(n_i) \leq_\Gamma S_i \ \text{ if } \mathrm{rc}_g(n_i) = \odot,$$
$$[\mathcal{T}(n_i)]_\Gamma \leq_\Gamma S_i \ \text{ if } \mathrm{rc}_g(n_i) = \otimes.$$

With a suitable adjustment of polymorphic uniqueness rule typings, the subject reduction theorem can be specified straightforwardly.

As to decidability, we will describe how to compute uniqueness variants of conventional typings.

Uniqueness Type Inference

In this section we will describe how to compute uniqueness variants of conventional typings.

Definition 31. Let $\mathcal{T}$ be a typing for g.
(i) A *g-attribution* of $\mathcal{T}$ is a uniqueness typing $\langle \mathcal{T}, \Gamma \rangle$ for g such that $|\mathcal{T}| = \mathcal{T}$.
(ii) Such an attribution is called *principal* if for any g-attribution $\langle \mathcal{T}', \Gamma' \rangle$ there exists an attribute substitution $\diamond$ with

$$\mathcal{T}' = \mathcal{T}^{\diamond} \qquad \text{and} \qquad \Gamma' \vdash \Gamma^{\diamond}.$$

Principal Attribution Theorem 32 *There exists a computable function* Attr *such that for each typing for g the following holds. If $\mathcal{T}$ has a g-attribution, then* $\mathrm{Attr}(g, \mathcal{T})$ *is a principal attribution; otherwise* $\mathrm{Attr}(g, \mathcal{T}) = \mathsf{fail}$.

For first order uniqueness typing it can be shown that if a conventional typing for a graph, say g, is attributable then g's principal typing is also attributable. Due to a subtle restriction on the coercion possibilities for the function space constructor, this does not hold anymore for the higher-order system (see Barendsen and Smetsers (1996)). This implies that the natural way for deriving uniqueness types, which tries to determine an attribution of the principal conventional typing may fail, whereas an instance of this principal typing is attributable. Consequently, there is no 'Principal Uniqueness Type Theorem' for higher order uniqueness typing. This is reflected in the description of the typing procedure in *Clean*.

Uniqueness Type Inference in Clean

In order to translate the above into a suitable actual typing algorithm we indeed try to lift principal typings. If this attempt fails, however, we do not try any specific instances but consider the graph untypable.

As a consequence, the underlying conventional typings of the derived uniqueness types are exactly the principal ones, so from the programmer's point of view the uniqueness system is a transparent extension of conventional typing: if one disregards the uniqueness information the types are as one would expect.

Having seen how to derive graph typings in a given environment, we can focus on type inference for rewrite rules. As in any other functional language, in *Clean* type checking is concerned with the determination of a suitable environment type for each function symbol, such that all program parts are well-typed.

This actually boils down to determining uniqueness types for the right-hand sides of the function definitions, using the above procedure. The only problem is the possibility of (mutually) recursive function specifications. It is well-known that typing of these definitions is undecidable in general.

In fact, the *Clean* compiler adopts the Hindley-Milner approach towards recursion: in the definition of, say, **F**, all occurrences of **F** should be typed with **F**'s environment type (i.e., without instantiation). Indirect recursion is treated similarly.

Alternative reference count analysis

A straightforward (static) reference counting treats all references to a given object in the same way. This can be refined: multiple access to a unique argument is harmless if one knows that only one of the references will be present at the moment of evaluation.

An example of this evaluation-strategy-aware (dynamic) reference counting is the treatment of conditional expressions in *Clean*. For example, if we define the conditional by

$$\begin{aligned}&\mathbf{Cond}(\mathbf{True}, x, y) \to x\\&\mathbf{Cond}(\mathbf{False}, x, y) \to y\end{aligned}$$

and compute $\mathbf{Cond}(b, g, g')$ in the standard way, the condition is evaluated first (with possible sharing between b and g, g') and subsequently one of the alternatives is chosen and evaluated (so sharing between g and g' has disappeared). This suggests that we can distinguish between references to the same object inside b, g and g' respectively, allowing a less restrictive uniqueness typing. In fact, the results of Barendsen and Smetsers (1993) already abstract from the way references are counted: they capture both the standard and the refined approach.

8 Conclusion

TGRS's have successfully been used as common language between theoreticians and implementors. Using TGRS's as computational model a state-of-the-art implementation has been obtained. We have used TGRS's not only as base for the implementation but also as a base for the semantics of Clean itself. This enabled us to introduce new important features into the language, such as uniqueness types (related to linear types).

Functions that have a "unique" object as argument are guaranteed to be applied with a non-shared object. Such a function therefore will have private access to this object such that the space occupied by the object can be reused to construct the function result. This means that one can perform destructive updates within a pure function language. This enables both writing pure functional programs which run almost as efficient as programs written in C as well as purely functional interfacing with imperative programs.

References

Ariola, Z.M. and J.W. Klop (1996). Equational Term Graph Gewriting, *Fundamentae Informaticae*, Vol. 26, Nrs. 3, 4, pp. 207–240.

Barendregt, H.P. (1984). *The Lambda Calculus: Its Syntax and Semantics*, Studies in Logic 103, second, revised edition, North-Holland, Amsterdam.

Barendregt, H.P., M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer and M.R. Sleep (1987). Term graph reduction, *in:* J.W. de Bakker, A.J. Nijman and P.C. Treleaven (eds.), *Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE)* II, Eindhoven, The Netherlands, Lecture Notes in Computer Science 259, Springer-Verlag, Berlin, pp. 141–158.

Barendsen, E. (1995). *Types and Computations in Lambda Calculi and Graph Rewrite Systems*, Dissertation, University of Nijmegen.

Barendsen, E. and J.E.W. Smetsers (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, Lecture Notes in Computer Science 761, Springer-Verlag, Berlin, pp. 41–51. Full paper: see Technical Report CSI-R9328, University of Nijmegen, and Barendsen (1995).

Barendsen, E. and J.E.W. Smetsers (1994). Extending graph rewriting with copying, *in:* H.J. Schneider and H. Ehrig (eds.), *Graph Transformations in Computer Science, International Workshop*, Dagstuhl Castle, Germany, Lecture Notes in Computer Science 776, Springer-Verlag, Berlin, pp. 51–70.

Barendsen, E. and J.E.W. Smetsers (1996). Uniqueness typing for functional languages with graph rewriting semantics, *Mathematical Structures in Computer Science*, Vol. 6, pp. 579–612.

Brus, T., M.C.J.D. van Eekelen, M.O. van Leer and M.J. Plasmeijer (1987). Clean: A language for functional graph rewriting, *Proceedings of the Conference on Functional Languages and Computer Architectures (FPCA)*, Portland, Oregon, Lecture Notes in Computer Science 274, Springer-Verlag, Berlin, pp. 364–384.

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.

Guzmán, J.C. and P. Hudak (1990). Single-threaded polymorphic lambda calculus, *Proceedings of the 5th Annual Symposium on Logic in Computer Science*, Philadelphia, IEEE Computer Society Press, pp. 333–343.

Hudak, P., P.L. Wadler, Arvind, B Boutel, J. Fairbairn, J. Fasel, K. Hammond, J. Hughes, T. Johnsson, R. Kieburtz, R.S. Nikhil, S.L. Peyton Jones, M. Reeve, D. Wise and J. Young (1990). Report on the functional programming language haskell, *Technical report*, Department of Computer Science, Glasgow University.

Klop, J.W. (1992). Term rewrite systems, *in:* S. Abramsky, D.M. Gabbay and T.S.E. Maibaum (eds.), *Handbook of Logic in Computer Science*, Vol. II, Oxford University Press.

Lamping, J. (1990). An algorithm for optimal lambda calculus reduction,, *Proc. of POPL'90: Seventeenth annual ACM symposium on Principles of Programming Languages*, San Francisco, California, pp. 16–30.

Milner, R. (1984). A proposal for standard ML, *Proceedings of the ACM Symposium on LISP and Functional Programming*, Austin, pp. 184–197.

Nöcker, E.G.J.M.H. (1993). Strictness analysis using abstract reduction, *Proc. of Conference on Functional Programming Languages and Computer Architecture (FPCA '93)*, Kopenhagen, ACM Press, pp. 255–265.

Nöcker, E.G.J.M.H. and J.E.W. Smetsers (1993). Partially strict non-recursive data types, *Journal of Functional Programming*.

Nöcker, E.G.J.M.H., J.E.W. Smetsers, M.C.J.D. van Eekelen and M.J. Plasmeijer (1991). Concurrent Clean, *in:* E.H.L. Aarts, J. van Leeuwen and M. Rem (eds.), *Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE)* II, Eindhoven, The Netherlands, Lecture Notes in Computer Science 505, Springer-Verlag, Berlin, pp. 202–219.

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

Smetsers, J.E.W. (1993). *Graph Rewriting and Functional Languages*, Dissertation, University of Nijmegen.

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

Turner, D.A. (1985). Miranda: A non-strict functional language with polymorphic types, *in:* J.P. Jouannaud (ed.), *Proceedings of the Conference on Functional Languages and Computer Architectures (FPCA)*, Nancy, France, Lecture Notes in Computer Science 201, Springer-Verlag, Berlin, pp. 1–16.

Turner, D.N., P. Wadler and C. Mossin (1995). Once upon a type, *Proceedings of the Conference on Functional Languages and Computer Architectures (FPCA)*, La Jolla, California, ACM Press, pp. 1–11.

Wadler, P. (1990). Linear types can change the world!, *Proceedings of the Working Conference on Programming Concepts and Methods*, Israel, North-Holland, Amsterdam, pp. 385–407.

Wadsworth, C.P. (1971). *Semantics and Pragmatics of the Lambda Calculus*, Dissertation, Oxford University.

This article was processed using the LaTeX macro package with LLNCS style